AFIPS

CONFERENCE PROCEEDINGS

VOLUME 41

PART I

1972

FALL JOINT COMPUTER CONFERENCE

December 5 - 7, 1972
Anaheim, California

AFIPS PRESS
210 SUMMIT AVENUE
MONTVALE, NEW JERSEY 07645

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1972 Fall Joint Computer Conference Committee or the American Federation of Information Processing Societies, Inc.

Library of Congress Catalog Card Number 55-44701

AFIPS PRESS
210 Summit Avenue
Montvale, New Jersey 07645

Printed in the United States of America

CONTENTS

PART I

OPERATING SYSTEMS

ARCHITECTURE FOR HIGH SYSTEM AVAILABILITY

COMPUTING INSTALLATIONS—PROBLEMS AND PRACTICES

COMPUTER GRAPHICS

SOFTWARE ENGINEERING—THEORY AND PRACTICE

(PART I)

SUPERCOMPUTERS—PRESENT AND FUTURE

MAINTENANCE AND SYSTEM INTEGRITY

COMPUTER SIMULATIONS OF COMPUTER SYSTEMS

SOFTWARE ENGINEERING—THEORY AND PRACTICE

(PART II)

ARCHITECTURE LIMITATIONS IN LARGE-SCALE COMPUTATION AND DATA PROCESSING

(Panel Discussion—No Papers in this Volume)

ARRAY LOGIC AND OTHER ADVANCED TECHNIQUES

ADVANCES IN SIMULATION

PRIVACY AND THE SECURITY OF DATABANK SYSTEMS

ARRAY LOGIC—WHERE ART THOU?

(Panel Discussion—No Papers in this Volume)

HARDWARE-FIRMWARE-SOFTWARE TRADE-OFFS

HUMAN ENGINEERING OF PROGRAMMING SYSTEMS—THE USER'S VIEW

DATA COMMUNICATION SYSTEMS

MEASUREMENT OF COMPUTER SYSTEMS—SYSTEM PERFORMANCE

(Panel Discussion—No Papers in this Volume)

MEMORY ORGANIZATION AND MANAGEMENT

DYNAMIC PROGRAM BEHAVIOR

COMPUTER ASSISTED EDUCATIONAL TEST CONSTRUCTION

Properties of disk scheduling policies in multiprogrammed computer systems

by TOBY J. TEOREY

University of Wisconsin
Madison, Wisconsin

INTRODUCTION

The subject of scheduling for movable head rotating storage devices, i.e., disk-like devices, has been discussed at length in recent literature. The early scheduling models were developed by Denning,[3] Frank,[6] and Weingarten.[14] Highly theoretical models have been set forth recently by Manocha,[9] and a comprehensive simulation study has been reported on by Teorey and Pinkerton.[12]

One of the goals of this study is to develop a model that can be compared with the simulation results over a similar broad range of input loading conditions. Such a model will have two advantages over simulation: the computing cost per data point will be much smaller, and the degree of uncertainty of a stable solution will be decreased.

Although the previous analytical results on disk scheduling are valid within their range of assumptions, they do not provide the systems designer with enough information to decide whether or not to implement disk scheduling at all; neither do they determine which scheduling policy to use for a given application, be it batch multiprogramming, time sharing, or real-time processing. The other goal of this study is to provide a basis upon which these questions can be answered.

The basic scheduling policies are summarized with brief descriptions in Table I. Many variations of these policies are possible, but in the interest of mathematical analysis and ease of software implementation we do not discuss them here.

SCAN was first discussed by Denning.[3] He assumed a mean (fixed) queue length and derived expected service time and mean response time. The number of requests in the queue was assumed to be much less than the number of cylinders, so the probability of more than one request at a cylinder was negligible. We do not restrict ourselves to such an assumption here. Improvements on the definition and representation of SCAN have been suggested by Coffman and Denning,[2] Manocha,[9] and Merten.[10] The implementation of SCAN is often referred to as LOOK,[10,12] but we retain the name SCAN for consistency within this paper. Both C-SCAN[9,11,12,13] and the N-step scan[6,12,13] have been discussed or studied previously and the Eschenbach scheme was developed for an airlines system.[14] Because it requires overhead for rotational optimization as well as seek time optimization it is not included in the following discussion. In the simulation study[12] it was seen that the C-SCAN policy, with rotational optimization, was more appropriate than the Eschenbach scheme for all loading conditions, so we only consider C-SCAN here.

The simulation results indicated the following, given that cylinder positions are addressed randomly:[12] under very light loading all policies perform no better than FCFS. Under medium to heavy loading the FCFS policy allowed the system to saturate and the SSTF policy had intolerable variances in response time. SCAN and the N-step policies were superior under light to medium loading, and C-SCAN was superior under heavy loading.

We first investigate various properties of the N-step scan, C-SCAN, and SCAN, since these are the highest performance policies that optimize on arm positioning time (seek time). The properties include mean, variance, and distribution of response time; and the distribution of the positions of requests serviced as a function of distance from the disk arm before it begins its next sweep. Response time mean and variance are then compared with simulation results.

A unified approach is applied to all three policies to obtain mean response time. The expressions are nonlinear and require an iterative technique for solution; however, we can easily show that sufficient conditions always exist for convergence.

Finally, we look at the factors that must be considered in deciding whether or not to implement disk

TABLE I—Basic Disk Scheduling Policies

1. FCFS (First-come-first-served): No reordering of the queue.
2. SSTF (Shortest-seek-time-first): Disk arm positions next at the request that minimizes arm movement.
3. SCAN: Disk arm sweeps back and forth across the disk surface, servicing all requests in its path. It changes direction only when there are no more requests to service in the current direction.
4. C-SCAN (Circular scan): Disk arm moves unidirectionally across the disk surface toward the inner track. When there are no more requests to service ahead of the arm it jumps back to service the request nearest the outer track and proceeds inward again.
5. N-step scan: Disk arm sweeps back and forth as in SCAN, but all requests that arrive during a sweep in one direction are batched and reordered for optimum service during the return sweep.
6. Eschenbach scheme: Disk arm movement is circular like C-SCAN, but with several important exceptions. Every cylinder is serviced for exactly one full track of information whether or not there is a request for that cylinder. Requests are reordered for service within a cylinder to take advantage of rotational position, but if two requests overlap sector positions within a cylinder, only one is serviced for the current sweep of the disk arm.

scheduling in a complex system. In practice, considerable attention should be given to these factors before thinking about which policy to use.

N-STEP SCAN

The N-step scan is the simplest scheduling policy to model using the approach discussed here. While the disk arm is sweeping across the surface to service the previous group of requests, new requests are ordered linearly for the return sweep. No limit is placed on the size of the batch, but at equilibrium we know the expected value of that size to be L, the mean queue length. Furthermore, we know that the resulting request position distribution will be the same as the input distribution, which we assume to be uniform across all the disk cylinders. We also assume the following:

1. Request interarrival times are generated from the exponential distribution.
2. File requests are for equal sized records. This simplifies the analysis. We assume that the total service time distribution (seek time plus rotational delay plus transmission) is general and cannot be described by any simple distribution function. We also assume that the access time (seek time plus rotational delay) dominates the total service time, so that fixed record size (constant transmission time) is a fair approximation for our purpose of a comparative analysis.
3. Only a single disk drive with a dedicated controller and channel is considered, and there is only one movable head per surface. All disk arms are attached to a single boom so they must move simultaneously. A single position of all the read/write heads defines a cylinder.
4. Seek time is a linear function of seek distance.
5. No distinction is made between READ and WRITE requests, and the overhead for scheduling is assumed negligible.

If there are L requests in the queue at equilibrium and C cylinders on the disk, we partition the disk surface into C_1 equal regions (as defined below) and assume that at least one request lies in the center of that region. This partition is only valid when seek time is a linear function of distance. C_1 is computed as follows: since the distribution of L requests serviced is uniform, the probability that cylinder k has no requests is given by

$$P_k = \left(1-\frac{1}{C}\right)^L \quad \text{for all } k \tag{1}$$

The expected number of cylinders with no requests is $C_0 = CP_k$, so that the expected number of cylinders requiring service is:

$$\begin{aligned} C_1 &= C - C_0 \\ &= C - C\left(1-\frac{1}{C}\right)^L \\ &= C\left[1-\left(\frac{C-1}{C}\right)^L\right] \end{aligned} \tag{2}$$

If the incoming requests are placed at random and the disk arm has equal probability of being at any cylinder, we know that the expected distance between an incoming request and the current position of the disk arm is approximately $C/3$ for large C. Typically, $C \geqq 200$ for currently available disks. In Figure 1 we see the possible paths taken from the disk arm to the new request for the expected distance of $C/3$. The expected number of requests serviced before the new request is serviced is L, and the mean response time is

$$W = LT_s = T_{sw} \tag{3}$$

where T_s is the expected service time per request and T_{sw} is the expected sweep time from one extreme of the disk surface to the other.

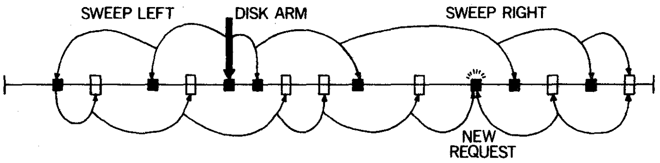


Figure 1

The expected service time under the assumptions listed above was derived by Teorey and Pinkerton[12] as follows:

$$T_s = P\left(T_{sk} + \frac{T}{2} + \frac{T}{m}\right)$$

$$+(1-P)\,\frac{T}{m}\left[\frac{(mt-2)(m-1)}{2(mt-1)}+1\right] \qquad (4)$$

where P is the probability that a seek is required to service the next request, T_{sk} is the expected seek time, T is the rotational time of a disk, m is the number of sectors per track, and t is the number of tracks per cylinder. Under our conditions, $P = C_1/L$, and we simplify expression (4) by making the following definition:

$$a = \frac{T}{m}\left[\frac{(mt-2)(m-1)}{2(mt-1)}+1\right] \qquad (5)$$

Also, for a linear seek time characteristic

$$T_{sk} = \begin{cases} T_{\min} + \dfrac{\Delta T}{C_1} & C_1 \geqq 3 \\[2ex] T_{\min} + \dfrac{\Delta T}{3} & C_1 < 3 \end{cases} \qquad (6)$$

where $\Delta T = T_{\max} - T_{\min}$, $T_{\min}$ is the seek time for a distance of 1 cylinder, and $T_{\max}$ is the seek time for a distance of $C-1$ cylinders. Restating (4) we now have

$$T_s = \frac{C_1}{L}\left(T_{\min} + \frac{\Delta T}{C_1} + \frac{T}{2} + \frac{T}{m}\right) + \left(1 - \frac{C_1}{L}\right)a \qquad (7)$$

At equilibrium the mean number of incoming requests that arrive in one complete sweep is L, because the departure rate and the arrival rate must be the same.

$$L = \lambda T_{sw} = \lambda L T_s \qquad (8)$$

where λ is the input (throughput or access) rate. Dividing both sides of (8) by L and substituting (7) we have:

$$1 = \lambda\left[\frac{C_1}{L}\left(T_{\min} + \frac{\Delta T}{C_1} + \frac{T}{2} + \frac{T}{m}\right) + \left(\frac{L - C_1}{L}\right)a\right]$$

$$L = \frac{\lambda C_1(T_{\min} + \Delta T/C_1 + T/2 + T/m - a)}{1 - \lambda a} \qquad (9)$$

Equation (9) computes mean queue length in terms of the input rate λ, the known disk hardware characteristics, and C_1. C_1 is, however, a nonlinear function of L. We solve (9) by estimating an initial value for L in (2) and iteratively substituting (2) into (9) until the process converges.

Convergence

Rewriting (9) in terms of (2) we obtain

$$L(1-\lambda a)$$

$$= \lambda\Delta T + \lambda C\left[1 - \left(\frac{C-1}{C}\right)^L\right]\left(T_{\min} + \frac{T}{2} + \frac{T}{m} - a\right)$$

$$L = \frac{\lambda \Delta T}{1-\lambda a} + \frac{\lambda C}{1-\lambda a}\,(T_{\min} + T/2 + T/m - a)$$

$$- \frac{\lambda C}{1-\lambda a}\,(T_{\min} + T/2 + T/m - a)\left(\frac{C-1}{C}\right)^L \qquad (10)$$

Letting $K_1 = \lambda\Delta T/(1-\lambda a) + [\lambda C/(1-\lambda a)](T_{\min} + T/2 + T/m - a)$ and $K_2 = [\lambda C/(1-\lambda a)]$. $(T_{\min} + T/2 + T/m - a)$ we obtain after i iterations:

$$L_{i+1} = K_1 - K_2\left(\frac{C-1}{C}\right)^{L_i} \qquad (11)$$

Assuming that $L_i > 0$ for all i, and $1 - \lambda a > 0$ (no saturation), we have:

$$\left.\begin{matrix} L_i > 0 \\ 1-\lambda a > 0 \end{matrix}\right\} \Rightarrow 0 \leqq \left(\frac{C-1}{C}\right)^{L_i} < 1$$

$$\Rightarrow 0 < K_1 - K_2 < L_{i+1} \leqq K_1 < \infty$$

[Boundedness on L_{i+1}]

From (11) we can easily see that

$$L_i > L_{i-1} \Rightarrow L_{i+1} > L_i \quad \text{and} \quad L_i < L_{i-1} \Rightarrow L_{i+1} < L_i.$$

[Monotonicity]

Since every bounded increasing (or decreasing) set of real numbers has a limit, (11) converges to L at equilibrium.

For this technique, each data point of L vs. λ requires less than one second of UNIVAC 1108 CPU time, whereas each point of the simulation requires over 30 seconds. Mean response time is obtained from Little's formula[8] and can be verified by resubstitution of L back through (3).

Under light loading conditions, i.e., when $L \ll C$, the probability that a seek is required for every request approaches 1. Under such conditions $C_1 \cong L$ and the following closed form expression is obtained:

$$L \cong \lambda \Delta T \Big/ \left[1-\lambda\left(T_{\min}+\frac{T}{2}+\frac{T}{m}\right)\right] \qquad (12)$$

Variance of response time

Simulation results verify the intuitive suggestion that the response time distribution for the N-step scan approaches the simple triangular distribution shown in Figure 2. If we partition the disk into only 10 or 20 regions, place the disk arm in each of those regions with equal probability, and then keep a cumulative total of the probabilities of response times at each point, we will obtain a discrete approximation of Figure 2. Accepting this approximation, variance is found by

$$\begin{aligned}\sigma_W{}^2 &= E(X^2)-[E(X)]^2\\ &= \int_0^{T_{sw}} x^2\left(\frac{x}{T_{sw}{}^2}\right)dx\\ &\quad+\int_{T_{sw}}^{2T_{sw}} x^2\left(\frac{2T_{sw}-x}{T_{sw}{}^2}\right)dx-(T_{sw})^2\\ &= T_{sw}{}^2/6\\ &= W^2/6 \qquad (13)\end{aligned}$$

Thus, the N-step scan provides a very low variance in response time.

C-SCAN

The C-SCAN policy is an attempt to decrease variance of response time without degrading the maximum possible throughput rate or increasing the mean response time.

We assume requests distributed uniformly over all cylinders. Since the disk arm always moves unidirectionally to service requests, the expected density of requests just ahead of the disk arm is uniform (provided we ignore the slight aberration of jumping back to the outermost request once per cycle). Figure 3 shows a simulation result of this distribution under light and heavy loading conditions. Consequently, the computation of expected service time T_s is the same for C-SCAN as it is for the N-step scan, i.e., equation (4), except the number of requests serviced per sweep is no longer restricted to L, but is some unknown quantity L'. Therefore we now have $C_1=C\{1-[(C-1)/C]^{L'}\}$, $P=C_1/L'$, and

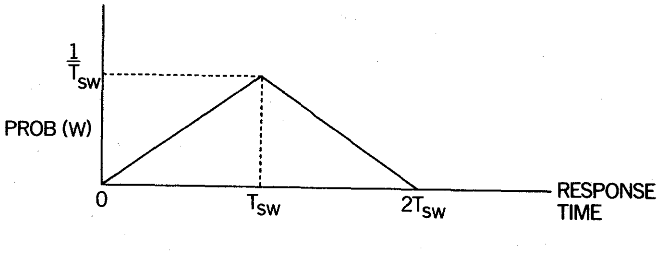


Figure 2

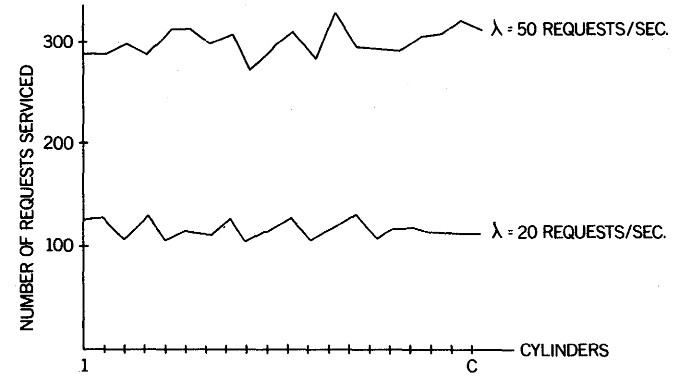


Figure 3

$$T_s=\frac{C_1}{L'}\left(T_{\min}+\frac{\Delta T}{C_1}+T/2+T/m\right)+(1-C_1/L')a \qquad (14)$$

Total time for one cycle of C-SCAN is the expected service time for L' requests. This includes C_1-1 seeks and a return seek to the outermost request, which is less than or equal to $T_{\max}$:

$$T_{sw}=L'T_s-(T_{\min}+\Delta T/C_1)+(T_{\max}-\Delta T/C_1) \qquad (15)$$

At equilibrium the number of incoming requests that arrive in one sweep (cycle) time is L', the total number of requests serviced:

$$\begin{aligned}L' &= \lambda T_{sw}\\ &= \lambda[C_1(T_{\min}+\Delta T/C_1+T/2+T/m)\\ &\quad +L'(1-C_1/L')a]-\lambda(T_{\min}+\Delta T/C_1)\\ &\quad +\lambda(T_{\max}-\Delta T/C_1)\\ L'(1-\lambda a) &= \lambda C_1(T_{\min}+T/2+T/m-a)\\ &\quad +\lambda(\Delta T-T_{\min}-\Delta T/C_1+T_{\max}-\Delta T/C_1)\\ L' &= \frac{\lambda C_1}{1-\lambda a}(T_{\min}+T/2+T/m-a)\\ &\quad +\frac{2\lambda\Delta T}{1-\lambda a}-\frac{2\lambda\Delta T}{(1-\lambda a)C_1} \qquad (16)\end{aligned}$$

Convergence

Letting $K_1=[\lambda C/(1-\lambda a)](T_{\min}+T/2+T/m-a)$ and $K_2=2\lambda\Delta T/(1-\lambda a)$ we can rewrite (16) after i iterations as

$$L'_{i+1}=K_1\left[1-\left(\frac{C-1}{C}\right)^{L_i'}\right] +K_2-\left\{K_2/C\left[1-\left(\frac{C-1}{C}\right)^{L_i'}\right]\right\} \qquad (17)$$

In order to derive sufficient conditions for convergence we assume the slightly stronger condition $L'_i>1$ for all i, and $1-\lambda a>0$ (no saturation).

$$\left.\begin{array}{l} L'_i>1\\ 1-\lambda a>0\end{array}\right\}\Rightarrow 0\leqq\left(\frac{C-1}{C}\right)^{L_i'}<\frac{C-1}{C}$$

$$\Rightarrow 1/C<1-\left(\frac{C-1}{C}\right)^{L_i'}\leqq 1$$

$$\Rightarrow 0<\frac{K_1}{C}<L'_{i+1}\leqq K_1+K_2-\frac{K_2}{C}<\infty$$

[Boundedness]

From (17) we see that the conditions for monotonicity of L' hold, and therefore the process converges.

Mean response time

The expected distance between the current arm position and a new request is approximately $C/3$ since, as with the N-step scan, the incoming requests are located at random, and the disk arm is at each cylinder with equal probability. An example of expected distance between a new request and the disk arm for C-SCAN is shown in Figure 4. Two possibilities occur with equal probability, as shown.

$$\begin{aligned} W &= \text{probability }\{\text{new request to the left}\}\cdot T_{sw}\{\text{left}\}\\ &\quad +\text{probability }\{\text{new request to the right}\}\\ &\quad \cdot T_{sw}\{\text{right}\}\\ &= \tfrac{1}{2}[\tfrac{2}{3}L'T_s-(T_{\min}+\Delta T/C_1)+(T_{\max}-\Delta T/C_1)]\\ &\quad +\tfrac{1}{2}[\tfrac{1}{3}L'T_s]\\ &= \tfrac{1}{2}(L'T_s-2\Delta T/C_1+\Delta T)\\ &= T_{sw}/2 \end{aligned} \qquad (18)$$

In other words the mean response time is one-half the expected sweep time. The mean queue length,

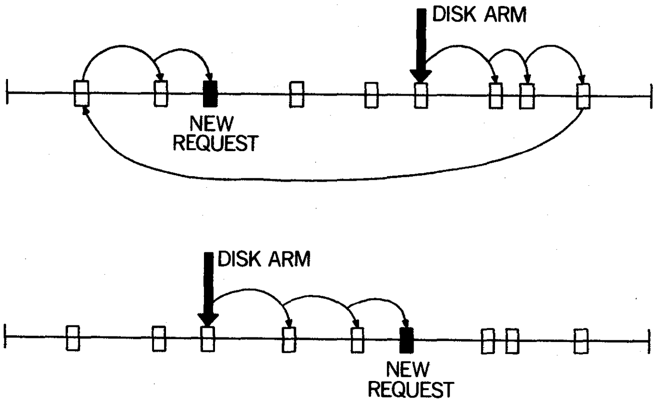


Figure 4

including the request in service, is

$$L=\lambda W=\tfrac{1}{2}\lambda T_{sw} \qquad (19)$$

but since $L'=\lambda T_{sw}$ we have

$$L'=2L \qquad (20)$$

which indicates that in one cycle the C-SCAN policy services twice as many requests as there are in the queue, and therefore should be able to attain a much higher throughput rate than the N-step scan.

Variance of response time

Because C-SCAN is a policy for a unidirectional disk arm, the distribution of response time is uniform between T/m (which we approximate to 0) and T_{sw}. The mean response time (18) is $T_{sw}/2$. For a uniform distribution the variance is given by

$$\sigma_W{}^2=\frac{(\tfrac{1}{2}T_{sw})^2}{3}=\frac{W^2}{3} \qquad (21)$$

which is twice the variance of the N-step scan.

SCAN

The SCAN access method has been the basic model for many implementations of scheduling in real systems. However, its properties are more complex than either of the other policies studied here. In order to determine the distribution of requests serviced as a function of distance from the extreme points of the disk, a simulation was devised and tested for very large samples under both light and heavy loading conditions. The results are summarized in Figure 5. They indicate in both cases that the number of requests per cylinder is a linear function of distance from the starting point of a sweep. This provides a basis for the linearity assumption in the analytical model that follows.

The expected distance between the current arm position and a new request is still approximately $C/3$ because the incoming requests are placed randomly, and for each full cycle the probability that the disk arm is at cylinder k is constant for all k. In Figure 6 we depict the linear distribution of request positions for the case $K_r < K_a$, where K_r is the cylinder position of a new request and K_a is the cylinder position of the disk arm. The possibility that $K_r > K_a$ also exists; each has a probability of .5.

1. $K_r < K_a$

l_L = number of requests serviced from K_a to C to K_r

$$= \text{Area } 3 + \text{Area } 1 + \text{Area } 2$$

$$= L' \qquad (22)$$

2. $K_r > K_a$

l_R = number of requests serviced from K_r to K_a

$$= \text{Area } 2$$

$$= \tfrac{1}{2}(K_r - K_a)\left(\frac{\substack{K-1\\ r}}{C-1}\cdot\frac{2L'}{C} + \frac{\substack{K-1\\ a}}{C-1}\cdot\frac{2L'}{C}\right)$$

$$= \frac{(K_r-K_a)(K_r+K_a-2)L'}{C(C-1)} \qquad (23)$$

To compute the expected number of cylinders with no requests, we first determine the probability of a given cylinder k obtaining the next incoming request:

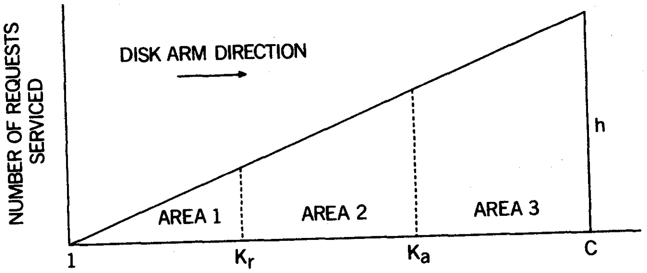


Figure 6

$$P_k = \frac{k-1}{C-1}\cdot\frac{2L'}{C}\Big/ l_R$$

for Area 2, $K_a \leqq k \leqq K_r$

$$P_k = \frac{k-1}{C-1}\cdot\frac{2L'}{C}\Big/ L' = \frac{k-1}{C-1}\cdot\frac{2}{C}$$

for Areas 1, 2, 3; $1 \leqq k \leqq C$ (23)

The input distribution is uniform; therefore each arrival of a new request represents a repeated Bernoulli trial of the same experiment. The probability that cylinder k remains empty is

$$P_{k0} = \left[1 - \left(\frac{k-1}{C-1}\cdot\frac{2L'}{C}\Big/ l_R\right)\right]^{l_R} \quad \text{for Area 2}$$

$$= \left(1 - \frac{k-1}{C-1}\cdot\frac{2}{C}\right)^{L'} \quad \text{for Areas 1, 2, 3}$$

(24)

and the expected number of occupied cylinders in that region is

$$C_2 = C/3 - \sum_{k=K_a}^{K_r}\left[1 - \left(\frac{k-1}{C-1}\cdot\frac{2L'}{C}\Big/ l_R\right)\right]^{l_R}$$

for Area 2

$$C_1 = C - \sum_{k=1}^{C}\left(1 - \frac{k-1}{C-1}\cdot\frac{2}{C}\right)^{L'} \quad \text{for Areas 1, 2, 3} \qquad (25)$$

Mean response time

The mean response time is given by

$$W = \text{Probability } \{K_r > K_a\}\cdot T_{sw}\{\text{Area } 2\}$$
$$+ \text{Probability } \{K_r < K_a\}\cdot T_{sw}\{\text{Areas } 1, 2, 3\}$$
$$= \tfrac{1}{2}[C_2(T_{\min} + \Delta T/C_2 + T/2 + T/m) + (l_R - C_2)a]$$
$$+ \tfrac{1}{2}[C_1(T_{\min} + \Delta T/C_1 + T/2 + T/m) + (L' - C_1)a]$$

(26)

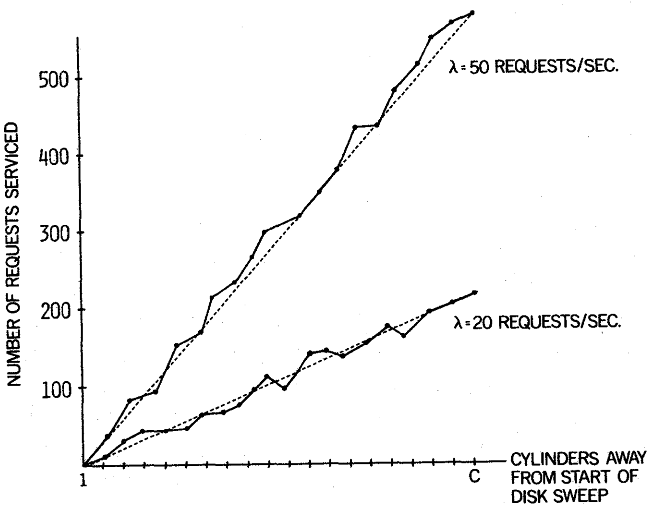


Figure 5

At equilibrium L' requests arrive in the time required for one complete sweep:

$$L' = \lambda T_{sw}$$

$$= \lambda[C_1(T_{\min} + \Delta T/C_1 + T/2 + T/m) + (L' - C_1)a]$$

$$= \frac{\lambda C_1(T_{\min} + \Delta T/C_1 + T/2 + T/m - a)}{1 - \lambda a} \qquad (27)$$

This expression is the same as (9) for the N-step scan except for the meaning of L' and C_1. Solution of (27) is obtained by iteration.

Convergence

Sufficient conditions for convergence of the above procedure for SCAN are $L'_0 > 0$ and $1 - \lambda a > 0$. The proof proceeds as before: Letting $K_1 = (\lambda/1 - \lambda a)[\Delta T + C(T_{\min} + T/2 + T/m - a)]$ and $K_2 = (\lambda/1 - \lambda a)[T_{\min} + T/2 + T/m - a]$ we can substitute (25) into (27) and obtain after i iterations:

TABLE II—Ratio of Requests Serviced per Sweep to Mean Queue Length for SCAN

Requests/second	L'/L
10	1.18
20	1.36
30	1.46
40	1.47
50	1.48
60	1.49
Limit	1.50

$$L'_{i+1} = K_1 - K_2 \sum_{k=1}^{C} \left(1 - \frac{2}{C} \cdot \frac{k-1}{C-1}\right)^{L_i'} \qquad (28)$$

$$\left.\begin{matrix} L'_i > 0 \\ 1 - \lambda a > 0 \end{matrix}\right\} \Rightarrow 0 \leqq \left(1 - \frac{2}{C} \cdot \frac{k-1}{C-1}\right)^{L_i'} < 1 \quad \text{for all } k \leqq C$$

$$\Rightarrow 0 \leqq \sum_{k=1}^{C} \left(1 - \frac{2}{C} \cdot \frac{k-1}{C-1}\right)^{L_i'} < C$$

$$\Rightarrow K_1 - K_2 C < L'_{i+1} \leqq K_1$$

$$[\text{Boundedness on } L'_{i+1}] \qquad (29)$$

From (28) we see that monotonicity of L' holds, and therefore the process converges.

The relationship between L and L' is dependent upon rate λ [see (26) and (27)]. For the characteristics of the IBM 2314 disk the following table illustrates this dependence.

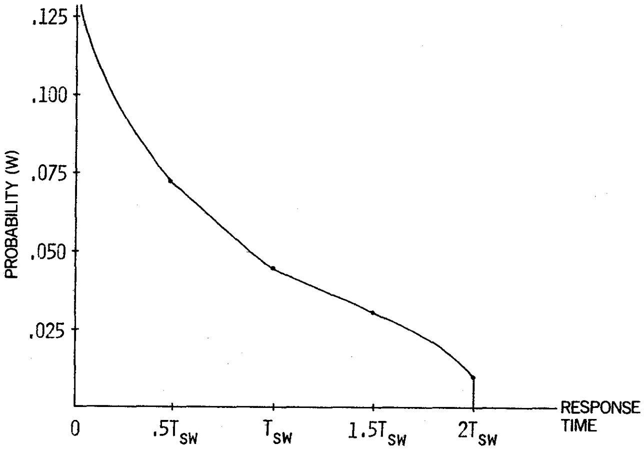


Figure 7

Variance of response time

The response time distribution for SCAN is not intuitively obvious. In order to obtain a close approximation to this distribution we can sample all possible

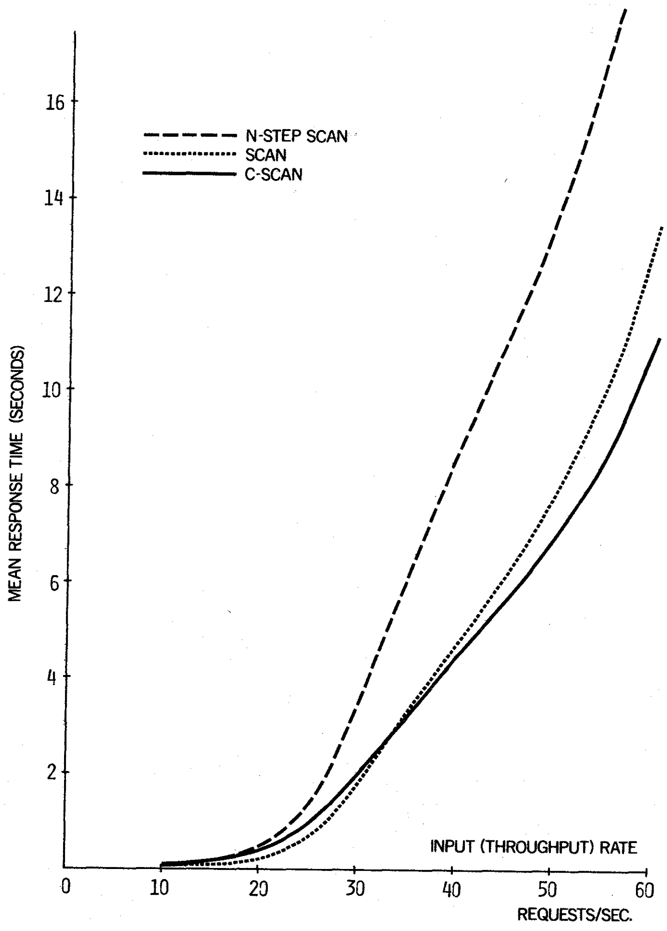


Figure 8

TABLE III—Properties of Disk Scheduling Policies

Property	N-step scan	C-SCAN	SCAN
Distribution of request locations	uniform	uniform	linear
Ratio L′/L (analytical)	1.0	2.0	1.5 (limit)
Ratio L′/L (simulation)	1.0	2.15	1.53
σ_W^2/W^2 (analytical)	.17	.33	.60
σ_W^2/W^2 (simulation)	.20	.35	.51

combinations of disk arm and new request positions. Given C cylinders, there are C^2 combinations of K_a and K_r positions. For each combination we can approximate the mean response time in terms of the expected number of requests (l_R or l_L) serviced between the two designated positions. From the resulting distribution (see Figure 7) the mean and variance of response time can be computed. We find that $W=.662\ T_{sw}$ and $\sigma_W{}^2=.264\ T_{sw}{}^2$. In the limit as λ becomes very large (but still below saturation) $W=.667\ T_{sw}$ from (26), os the two approximations are consistent.

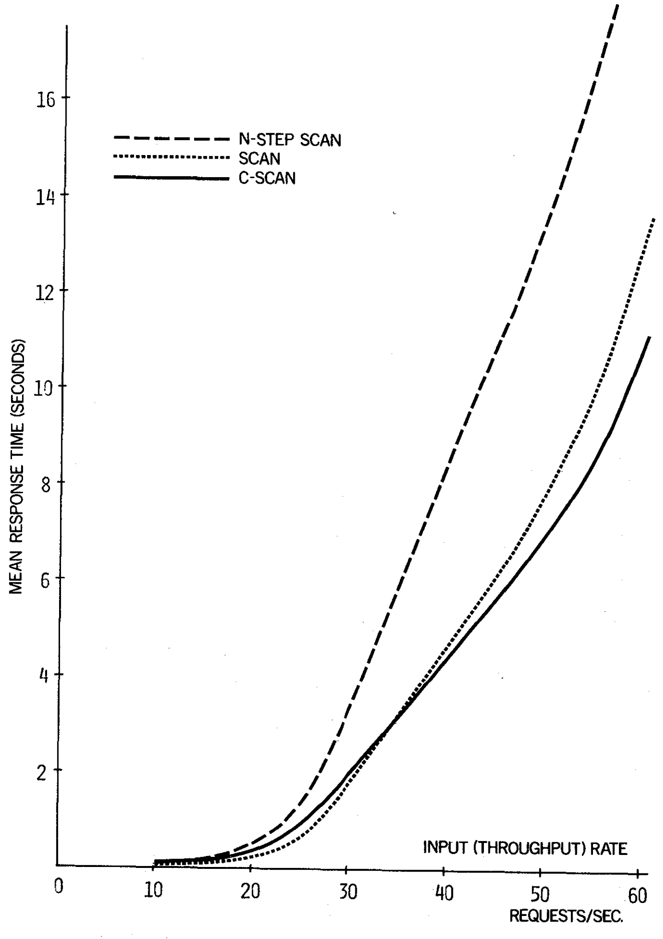


Figure 9

COMPARISON OF SCHEDULING POLICIES

The properties of the N-step scan, C-SCAN and SCAN are summarized below:

Mean response time is plotted in Figure 8 and Figure 9 as a function of input rate for the three high performance policies. The analytical results (Figure 8) correlate very closely with the simulation (Figure 9). Both results show a crossover occurring between C-SCAN and SCAN at approximately $\lambda=33$. The higher performance of C-SCAN at heavy loading appears to be the result of a uniform high density of requests always in front of the disk arm position. For $\lambda \leqq 20$ there is very little difference among these policies, and for $\lambda \leqq 10$ they all converge to the FCFS policy.

OPERATING SYSTEM AND HARDWARE CONSIDERATIONS

The analysis of scheduling policies has been thus far based on rather ideal mathematical conditions. As more practical limitations are modeled, the relative effectiveness of implementing disk scheduling compared to using only FCFS will in most cases decrease, reflecting real situations. Potentially, however, scheduling can be of more benefit if it is included as an integral component of an overall file system design rather than being treated as an independent algorithm. Let us now consider the following list of major factors that influence scheduling effectiveness:

1. Disk storage as the limiting resource in a large multiprogramming system.
2. Level of multiprogramming specified.
3. Multiple disk subsystems.
4. Nonuniform request distributions.
5. File organization techniques.
6. Seek time not dominant in total service time.

Limiting resource

In unbalanced multiprogramming systems, where congestion is not caused by disk storage, disk scheduling techniques should not be strongly considered. Instead, effort should be concentrated on optimizing or replacing the component causing poor system performance. Global decisions such as this must be made before individual components are to be upgraded, because a saturated device or subsystem determines the performance of the entire system. In a more balanced system other factors must be considered in relation to scheduling. When disk storage can be the cause of bottlenecks, scheduling should be included as a means of increasing throughput. An investigation of the effect scheduling has on overall system performance under such circumstances has been made by Teorey.[13]

Level of multiprogramming

A common misconception is that the level of multiprogramming is an upper bound on the queue length (L) at any system component. However, when an operating system breaks a program into multiple tasks or activities, and these are allowed to do I/O asynchronously, one obtains much longer queue lengths. (For example, consider a design which allows a distinct process for every input or output activity on every separate file opened by any user program.) For this reason we must not rule out the possibility of scheduling for batch systems with low levels of multiprogramming.

Disk activity typically varies quite considerably from device to device; consequently it may be necessary to measure the workload on each device to determine when scheduling should be used. When $L \leqq 3$ for an individual disk, the FCFS policy should be used. Certainly the level of multi-tasking will be an upper bound on the queue length of any one device, and when several devices are available the workload will probably be even less for any given one.

Typically batch systems operate at a level of 5 to 15 simultaneously executing programs. (The UNIVAC 1108 at the University of Wisconsin operates at a level of 9.) Time sharing systems may handle as many as 64 or 128 terminals; and in more specialized message handling systems several hundred or a thousand requests could be enqueued at any given time. Obviously, then, the potential for using scheduling to improve throughput is greatest in the latter type of system, but we must be aware that increased efficiency is usually achieved at the expense of mean and variance of individual response time. Such constraints in real-time systems must be seriously considered when selecting a scheduling policy.

Multiple disk facilities

Multiple device configurations have two main effects on disk performance. First, if requests are assumed uniformly distributed among the devices, the demand for an individual device is greatly reduced. Second, many (e.g., 8) devices may be serviced by a single controller, and many more (24 is not uncommon) may be serviced by a single channel. Consequently, control unit or channel saturation may be the cause of poor performance, despite individual disk drive efficiency. Theoretical models for multiple disk systems have been developed elsewhere.[1,5,8,11]

A new feature, rotational position sensing (RPS), is a disk hardware capability that allows the channel to be released during most of the rotational delay as well as the seek time delay, thus increasing its availability for overlapped operations. An analytical model for a multiple disk subsystem with RPS has been developed recently.[13] Multiple disk facilities without RPS have achieved effective masking of seek time due to concurrent arm positioning and heavy channel utilization. Consequently, disk arm scheduling has been of marginal benefit for such systems. However, because RPS decreases channel utilization it also decreases the degree of seek overlap, which in turn increases the potential effectiveness of scheduling. For example, an IBM 3330 disk system was analyzed with 4 and 8 drives, mean record sizes of 1.6K bytes and 8K bytes, with and without RPS, and with FCFS and C-SCAN scheduling.[13] The greatest throughput increase due to C-SCAN over FCFS (53%) occurred for 1.6K byte records, 4 drives, and RPS. Channel congestion, which works against the effectiveness of disk scheduling, is increased by using larger record sizes, adding more devices per channel, or by removing the RPS feature.

Nonuniform request distributions

Although a uniform request distribution does not typify general purpose batch systems, the actual distribution is highly dependent on installation workload and cannot be generalized. Some causes of nonuniform distributions are the use of physical devices dedicated to a single program (e.g., removable disk packs), priorities for disk service, and placement of the most highly used files or directories on a few contiguous cylinders, usually near the central disk arm position. Various estimates for nonuniform distributions have been investigated in other studies.[1,5,8]

These techniques tend to reduce the effectiveness of scheduling, and in some cases could be used in lieu of it. If scheduling is necessary in addition to systematically altering request distributions, the proper choice of an algorithm would depend on the amount of disk arm activity under these conditions. As with the uniform distribution, SCAN is preferred for light to medium loading and C-SCAN is preferred for heavy loading. At least for unimodal nonuniform distributions the most efficient algorithm still appears to be a simple scanning technique. In addition, if a few cylinders contain many requests, rotational optimization should be implemented as well as disk arm scheduling.

File organization techniques

Standard packages are available for various types of file organizations: sequential, calculated (hashing, scatter storage), tabular (index sequential), and others. A common characteristic of these techniques is that they require multiple accesses to the disk to obtain a single data record.

The index sequential access method (ISAM) requires access to a master index, a cylinder index, and then to the data record itself. The method is analyzed for a multiple disk facility by Seaman, et al.[11] They consider all accesses to the disk to obtain a single record as consecutive requests, that is, control of the disk arm is maintained until the record itself is finally accessed. Thus, in the worst case three consecutive random accesses could be made to obtain a single record. Normally, however, the master index is located in main storage, and under special conditions the cylinder index could be as well. In the latter case the record search is reduced to a single access, but at the expense of a large portion of main storage bound to a static index file. In the former case we have two accesses, but if part of one disk is dedicated to cylinder indexes the seek time for the index search is restricted to values near $T_{\min}$. Furthermore, we can overlap the next cylinder index search with the current record search. The two accesses are always on different modules and each can be scheduled independently.

Seek time not dominant

There are several other ways that diminish the effect of scheduling because the ratio of seek time to total service time is reduced. We note that scheduling of disk arm movement is merely a method to reduce seek time, and it can only have a significant effect on total service time if the seek time is the dominant factor. An upper bound on this dominance is established by the physical characteristics of the device. Some examples are provided in Table IV:

TABLE IV—T_{sk}/T_s for a Single Record Operation

	T_{sk}/T_s		
Device	Read or write 1 word	Read or write 1 track	Write & verify 1 track
IBM 2314	.83	.62	.49
IBM 3330	.78	.55	.42
UNIVAC FASTRAND II	.62	.35	.25

The Fastrand is limited by a very long rotation time, and is particularly slow for large record transfers which are typical for checkpoints, diagnostic dumps, and sorting. Further reductions in seek time dominance are caused by multi-phase operations such as "write and verify," retries for data read/write errors (hardware unreliability), and delays due to *I/O* channel busy.

SUMMARY

Disk scheduling should be implemented only after a careful consideration of the hardware configuration, the workload, and the type of operating system determines that the system would operate more efficiently. Selection of the best disk scheduling policy depends on the nature of the disk workload and the desired performance criteria of the particular application, i.e., throughput, mean response time, and/or variance of response time.

ACKNOWLEDGMENTS

I am deeply indebted to Tad Pinkerton and Bob Fitzwater for their helpful criticisms and suggestions.

APPENDIX

The following variables are frequently used throughout this analysis:

C	number of cylinders per disk.
C_0	expected number of cylinders with no requests
C_1	expected number of cylinders with at least one request.
λ	input (throughput) rate.
L	mean queue length including the one in service.
L'	expected number of requests serviced per sweep.
m	number of sectors per track.
P	probability that a seek will be required to service the next request.
σ_W^2	variance of response time.
t	number of tracks per cylinder.
$T_{\min}$	time to seek one cylinder.
$T_{\max}$	time to seek $C-1$ cylinders
ΔT	$T_{\max}-T_{\min}$.
T	disk rotation time.
T_s	expected service time.
T_{sw}	expected sweep time.
W	mean response time.

REFERENCES

1 J ABATE H DUBNER S B WEINBERG
Queueing analysis of the IBM 2314 disk storage facility
J ACM Vol 15 No 4 1968 pp 577-589

2 E G COFFMAN JR P J DENNING
Operating systems theory
Prentice-Hall Inc Englewood Cliffs N J 1972

3 P J DENNING
Effects of scheduling on file memory operations
Proc AFIPS 1967 SJCC Vol 30 pp 9-21

4 W FELLER
An introduction to probability theory and its applications
John Wiley and Sons Inc New York Vol 1 Third Edition 1968 pp 101-106

5 D W FIFE J L SMITH
Transmission capacity of disk storage systems with concurrent arm positioning
IEEE Trans on Computers EC-14 Aug 1965 pp 575-582

6 H FRANK
Analysis and optimization of disk storage devices for time-sharing systems
J ACM Vol 16 No 4 1969 pp 602-620

7 J D C LITTLE
A proof for the queuirg formula: L = λW
Opns Res Vol 9 No 3 1961 pp 383-387

8 G H MACEWEN
Performance of movable-head disk storage devices
Tech Rep No 72-4 Queens Univ Kingston Ontario Canada Jan 1972

9 T MANOCHA
Ordered motion for direct-access devices
SIAM 1971 Fall Meeting Madison Wisconsin Oct 11-13 1971

10 A G MERTEN
Some quantitative techniques for file organization
PhD Thesis Tech Rep No 15 Univ of Wisconsin Computing Center 1970

11 P H SEAMAN R A LIND T L WILSON
An analysis of auxiliary storage activity
IBM Syst J Vol 5 No 3 1966 pp 158-170

12 T J TEOREY T B PINKERTON
A comparative analysis of disk scheduling policies
Comm ACM Vol 15 No 3 1972 pp 177-184

13 T J TEOREY
The role of disk scheduling in multiprogrammed computer systems
PhD Thesis Univ of Wisconsin 1972 Madison Academic Computing Center Tech Rep

14 A WEINGARTEN
The analytical design of real-time disk systems
Proceedings IFIP Congr 1968 pp D131-D137

The interaction of multi-programming job scheduling and CPU scheduling

by J. C. BROWNE and JEAN LAN

The University of Texas at Austin
Austin, Texas

and

FOREST BASKETT

Stanford University
Palo Alto, California

INTRODUCTION

There have been very few systematic studies of the effect on system performance of strategies for scheduling jobs for execution in a multi-programming system.[1] Most of this work has been concerned with empirical efforts to obtain job mixes which effectively utilize the central processor.[2,3,4] These efforts are frequently carried out in commercial or production oriented installations where the job load consists of a relatively few jobs whose internal characteristics can be well determined. This approach is not feasible in an environment where internal job characteristics are not known before run time, or where internal job characteristics may vary rapidly. Such circumstances are often the case in an industrial or research laboratory or in a university computer center. This study uses as its measures for determining job scheduling strategies such quantities as are frequently known or can be accurately estimated such as amount of core memory required, processor service time required, etc. The specific job scheduling strategies used include first-come-first-serve (FCFS), shortest processor service time first (STF), smallest cost (cost = core size × processor service time) first (SCF), and smallest memory requirement first (SMF). We evaluated both preemptive resume and non-preemptive job scheduling. It is typical of virtually all of the previous work that the emphasis has been on improving CPU utilization. There may often be other goals which are more useful measures of performance such as throughput (job completion rate per unit time), the expected wait time before completion of a given class of job, the utilization of I/O resources, etc. We collected several measures of system performance including all of those listed previously to assess the effects of job scheduling. There has been very little previous study of the interaction between job scheduling and CPU scheduling. We systematically vary CPU scheduling algorithms in conjunction with alteration of job scheduling strategies. Those job scheduling strategies which give high throughput are characteristically observed to be more sensitive to CPU scheduling methods than those which yield relatively low throughput. We do not, however, attempt to correlate job scheduling methods with internal job characteristics such as CPU burst time, etc. We did, however, consider the effect of skewed CPU burst time distribution on performance under different pairs of strategies.

THE SYSTEMS MODEL

The model system which we simulate is based upon Control Data Corporation's (CDC) 6600 system at the University of Texas at Austin under the operation of the UT-1 and UT-2 operating systems. The CDC 6600 computer is a system of one very fast central processor (CPU), 10 peripheral processors (PP), and 12 data channels. The reader not familiar with the CDC 6000 series system is referred to Thornton[5] or the standard CDC reference manuals.[6] The UT-Austin 6600 system has 128K (K = 1,024) words of central core memory, 505,204 words of extended core storage (ECS), and 4 six million word disks (6638 disks). The principal features of the system are included in the model, the central processor: 85,000 words of central core memory (the balance is used by the operating system): the extended core storage, and the four disk channels: under

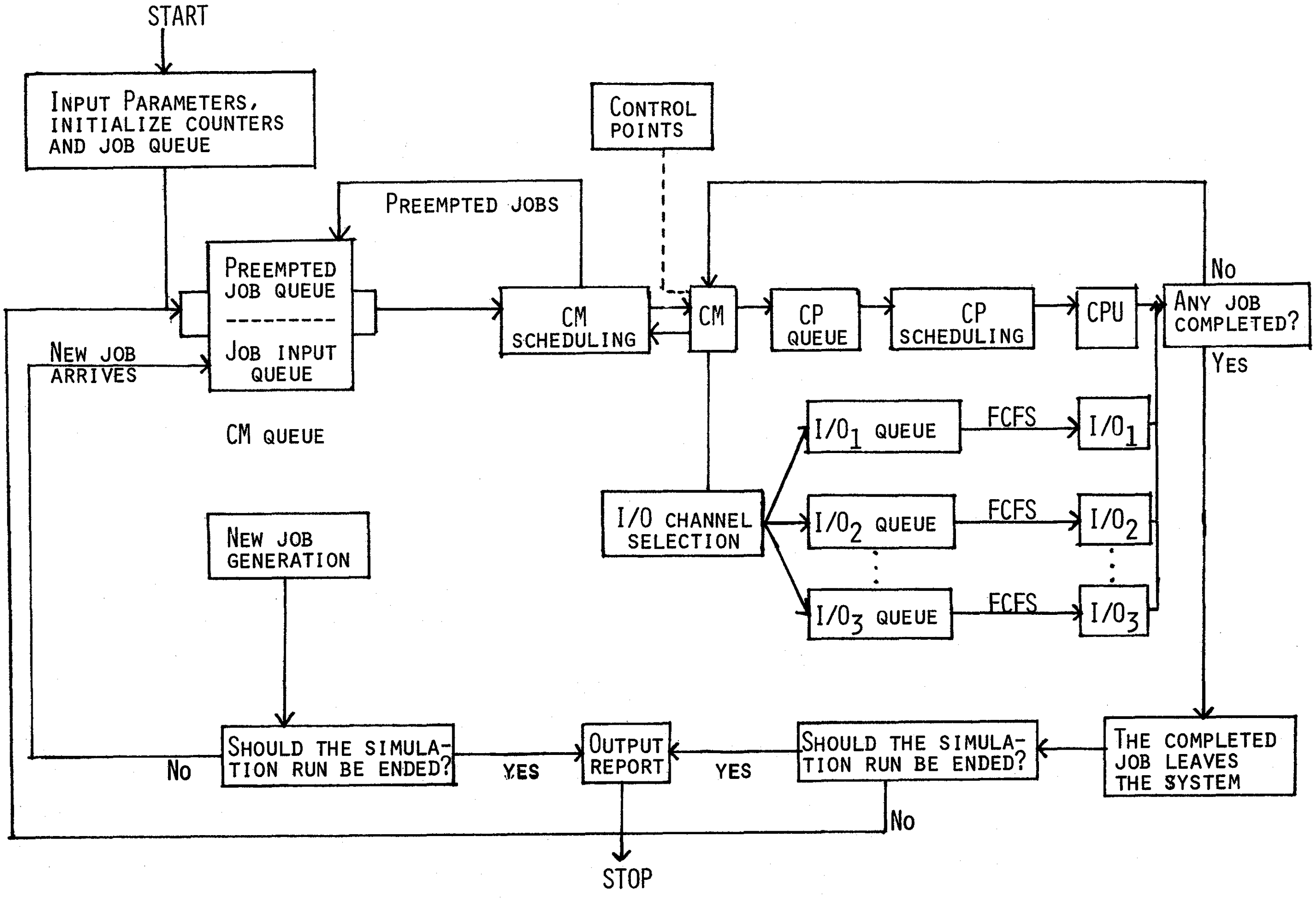


Figure 1—The computer system simulation model

UT-1 operation PP's were a surplus resource and could be left out of the model without materially affecting performance analysis. The operating systems under which the measurements were taken to parameterize this simulation model were the UT-1 and UT-2 operating systems. These operating systems are locally written. UT-1 used one PP as the system monitor (MTR); it was responsible for the coordination of all system activity. The 85,000 words of central memory available to user programs are allocated (by software) to seven (or fewer) control points which are virtual central processors. The multi-programming batch portion of UT-2 does not differ materially from UT-1 except for the allowance of up to 16 control points. A more complete description of the UT-1 system can be found in Schwetman[7] or Baskett, Raike and Browne.[1]

Both UT-1 and UT-2 have extensive measurement packages embedded in them [see Schwetman (7)]. The output of this measurement package is the source of the data which is used to parameterize the simulation model. Comparison of the output of the simulation model for key measures such as CPU utilization and channel utilization are used to validate the model. Figure 1 is a schematic diagram of the system model. The general operation of the model proceeds as follows: Ten jobs with specified storage requirements and central processor service times are generated and placed in the input queue. Jobs are selected from this input queue and operation of the system is started. The CPU burst times are selected from a specified (see following) distribution independently for each burst. The I/O burst times are similarly chosen from an exponential distribution. Channel selection is by a non-uniform discrete distribution for each I/O service request. The simulation proceeds with new jobs arriving at the input queue with an average interval of two seconds. The simulation run proceeds until 180 seconds of real ("clock-on-the-wall") time have passed. The simulation is then restarted nine successive times. The result of ten runs of 180 seconds are averaged to find average values and standard deviations for the performance measures. This procedure appears to be more reliable in terms of generating reproducible results than running the simulator for longer intervals. The complete set of simulations was run with exponential and hyperexponential CPU burst time distributions. Distribution functions for the memory requirements, total CPU service time required, arrival times, CPU burst time, I/O burst time, and channel selection are constructed from measurements made on the actual running system. For the CPU burst time, I/O burst time, and job arrival rate, analytic fits to the data were used. For storage requirements, total running time, and channel selection, table look-up procedures are used to generate a representation of the data distribution. The mean of the CPU mean burst time distribution was 48 ms. For the hyperexponential distribution a variance of 10 was used. A mean I/O burst time of 46 ms was taken from the measured data. The CM requirements were generated from a table which yields an approximate mean of 21,000 60-bit words. Channel 0 had a probability of selection of ½, channels 1, 2, and 3 each had probability of 1/6. The variance of the measured CPU burst time distribution was larger than 10. However, a variance as large as 10 captured the key features of the skewness of the distribution while still allowing a stable simulation. Larger variances (eg., 40) did not materially alter the performance measures but required very lengthy runs to reproduce the theoretical distributions. The job arrival rate was taken to have a mean of one every two seconds. This is the maximum rate observed in the system. The simulation program was written in FORTRAN; a thoroughly commented and flow charted version of this program is available on request. A more complete description of the simulation is given by Lan.[8]

VALIDATION

Since the simulation model is to be used to compare the relative merit of different scheduling algorithms rather than to predict absolute performance, the validation of interest is to be sure that the parameters put into the model reflect reasonably well the principal characteristics of the system and, more especially, the job mix. A good test, however, of how well the model captures the characteristics of the real system is to operate it using the scheduling algorithms used in the UT–1 operating system. Comparison to the real system can thus be obtained by examining the entries in the matrix of Table III with the data reported by Schwetman.[7] Schwetman reports central processor utilizations in the vicinity of 85 to 91 percent for various days production run. The average utilization of the four disk channels 10-57 percent, 1-20 percent, 2-19 percent, 3-18 percent, also fall well within the range observed by Schwetman for channel utilization. The actual numbers generated by the distribution functions were found to reproduce the theoretic means and variances of the CPU burst time, the I/O burst time, the channel selection, and the core size distribution function to less than ½ percent. This indicates a very high degree of stability and reproducibility in the simulated data. Another measured factor which can be compared is the average degree of multi-programming. We find 4.6 while Schwetman, including the remote terminal manager as a job as was appropriate for UT-1, measures 4.7. The neglected overhead in the central processor utilization is a known and small error under UT-1 where the central processor overhead was under 5 percent. The system monitor was a peripheral processor and monitor and service functions are done in the peripheral processors.

RESULTS OF VARIATION IN SCHEDULING ALGORITHMS

The goals of this simulation model are to evaluate the utility of several memory scheduling algorithms and several central processor scheduling algorithms and their interaction in terms of various measures of computer system performance. We studied the behavior of the model under four different memory scheduling algorithms.

(1) Shortest time to run first (STF)
(2) Smallest cost first (SCF)

In this context cost is defined to be the product of memory space required and central processor time required.

(3) Smallest memory first (SMF), in this algorithm one schedules the jobs according to the amount of central memory required.
(4) First-come-first-serve (FCFS), the classic discipline of queueing theory.

We considered both preemptive and non-preemptive memory scheduling. Table I compares preemptive and non-preemptive job scheduling for round-robin CPU scheduling. The central processor scheduling algorithms considered are:

(1) round-robin (RR) with an 8 millisecond (ms) quantum. Eight ms is the quantum size for the UT-1 and UT-2 operating systems. (A few runs were made with other quantum sizes.)

TABLE I—Results for Both Preemptive and Non-preemptive CM Scheduling Cases (With RR CPU Scheduling and Hyperexponential CPU Service Times)

Measures	Preemp. or non-preemp.	STF	SCF	SMF	FCFS
Number of jobs completed	P	76.5	75.1	37.4	25.4
	N	57.0	54.5	28.8	25.9
Degree of multiprogramming	P	4.362	4.601	5.610	4.430
	N	5.184	5.176	5.160	4.648
CM utilization	P	.944	.935	.888	.955
	N	.934	.927	.900	.941
CP utilization	P	.877	.889	.956	.940
	N	.938	.944	.959	.949
CP work time	P	158.9	161.3	173.4	170.3
	N	170.0	171.2	173.8	172.3
I/O work time	P	205.4	205.5	195.2	165.4
	N	190.6	186.0	175.7	169.0
Total CP and I/O work time	P	364.3	366.8	368.6	335.7
	N	360.6	357.2	349.6	341.3
I/O overlap time	P	167.8	171.3	180.8	145.5
	N	170.9	168.0	162.0	152.9
Average flow time for the completed jobs	P	11.8	11.6	15.4	58.8
	N	20.5	24.4	27.9	54.1
Average wait time for the completed jobs	P	7.8	7.8	10.8	51.0
	N	16.0	19.8	21.5	46.9
Total flow time	P	2797	2893	6198	7131
	N	4364	4847	6961	7142
Total wait time	P	2433	2526	5830	6796
	N	4003	4490	6612	6801
Average number of jobs in the system	P	16	16	34	39
	N	24	27	38	39
Number of swaps	P	99	92	27	9
	N	0	0	0	0

(2) Smallest time remaining (STR).
(3) Shortest burst time next (SBT).
(4) Longest burst time next (LBT).

The basic output of the simulation model is thus a set of 16 entries for each possible combination of scheduling disciplines for each measure of performance of interest. Table II is the matrix of entries for the case of an exponential CPU service time distribution. Table III is the matrix of entries for a hyper-exponential service time distribution. First-come-first-serve CPU scheduling was also tried for the hyperexponential CPU burst distribution. Table III thus has sets of 20 entries rather than 16. Most of our discussions will be couched in terms of the entries in Table III since it is known that the hyper-exponential distribution of service times is characteristic of most large scale multiprogramming computer systems. In most cases, the conclusions on the influence of scheduling algorithms on performance measures are corroborated by the exponential case (Table II).

Table IV is a summary chart of the principal results of this study. The left column of Table IV is a list of measures of computer system performance, throughput in terms of number of jobs completed, degree of multi-programming, central memory utilization, central processor utilization, I/O processing utilization, average flow time for complete jobs, average wait time for completed jobs, and the number of memory swaps as a measure of overhead. The rows of Table IV are the combination of scheduling algorithms which yield the best result for the performance measure in the left column. For example, in terms of throughput, the best combination of scheduling disciplines is STF-RR fol-

TABLE II—Results for Models with Exponentially Distributed CPU Burst Times

Measures	CP Schedule \ CM Schedule	STF	SCF	SMF	FCFS
Number of jobs completed (throughout)	RR	74.6	74.3	36.1	24.7
	STR	76.0	75.9	40.6	27.7
	SBT	75.3	74.4	37.1	25.3
	LBT	74.5	73.5	35.6	24.4
Degree of multiprogramming	RR	4.432	4.619	5.698	4.436
	STR	4.334	4.609	5.382	4.368
	SBT	4.385	4.637	5.717	4.418
	LBT	4.452	4.680	5.741	4.454
CM utilization	RR	.947	.939	.887	.958
	STR	.944	.935	.886	.953
	SBT	.946	.939	.890	.955
	LBT	.950	.940	.889	.957
CP utilization	RR	.941	.956	.975	.947
	STR	.944	.958	.973	.948
	SBT	.971	.980	.993	.975
	LBT	.922	.925	.949	.919
CP work time	RR	170.7	173.4	176.9	171.9
	STR	171.3	173.8	176.6	172.0
	SBT	176.1	177.8	180.2	176.9
	LBT	167.2	167.8	172.2	166.8
I/O work time	RR	155.3	158.7	166.9	162.1
	STR	156.8	159.2	166.0	161.1
	SBT	162.1	164.0	171.2	167.0
	LBT	152.6	153.7	162.2	157.6
Total CP & I/O work time	RR	326.0	332.3	343.9	334.0
	STR	328.1	333.0	342.5	333.1
	SBT	338.2	341.9	351.4	343.9
	LBT	319.9	321.4	334.4	324.4
I/O overlap time	RR	139.4	145.1	158.9	145.5
	STR	141.1	145.9	157.1	144.6
	SBT	154.3	158.4	169.3	159.5
	LBT	127.1	129.1	143.7	130.3
Average flow time for the completed jobs	RR	12.1	12.7	16.3	56.9
	STR	10.5	11.0	13.3	53.2
	SBT	12.3	12.7	16.6	57.2
	LBT	12.4	13.1	16.3	56.9
Average wait time for the completed jobs	RR	8.5	9.0	11.7	49.0
	STR	6.7	7.1	8.1	45.1
	SBT	8.5	9.0	11.8	49.5
	LBT	8.8	9.5	11.9	49.4
Total flow time	RR	2988	3056	6278	7184
	STR	2776	2814	5763	6788
	SBT	2919	3022	6206	7112
	LBT	3020	3114	6334	7216
Total wait time	RR	2662	2724	5934	6850
	STR	2448	2481	5420	6455
	SBT	2580	2680	5854	6768
	LBT	2701	2793	5999	6892
Average no. of jobs in the system	RR	17	17	35	40
	STR	15	16	32	37
	SBT	16	17	34	39
	LBT	17	17	35	40
No. of swaps	RR	100	89	24	8
	STR	104	99	27	8
	SBT	99	90	27	8
	LBT	99	87	24	8

TABLE III—Results for Models with Hyperexponentially Distributed CUP Burst Times

Measures	CM Schedule CP Schedule	STF	SCF	SMF	FCFS
Number of jobs completed (throughout)	RR	76.5	75.1	37.4	25.4
	STR	75.4	75.1	37.9	23.6
	SBT	75.6	74.9	35.9	21.8
	LBT	74.6	74.0	34.4	20.7
	FCFS	74.9	73.5	34.0	20.3
Degree of multiprogramming	RR	4.362	4.601	5.610	4.430
	STR	4.340	4.572	5.592	4.410
	SBT	4.363	4.609	5.698	4.430
	LBT	4.425	4.615	5.820	4.498
	FCFS	4.469	4.643	5.873	4.515
CM utilization	RR	.944	.935	.888	.955
	STR	.945	.937	.892	.954
	SBT	.944	.940	.890	.955
	LBT	.947	.939	.889	.956
	FCFS	.950	.942	.891	.958
CP utilization	RR	.877	.889	.956	.939
	STR	815	.828	.884	.862
	SBT	.841	.860	921	.889
	LBT	.804	.802	.856	.851
	FCFS	.827	.834	.890	.869
CP work time	RR	158.9	161.3	173.4	170.3
	STR	147.8	150.2	160.5	156.4
	SBT	152.5	155.8	167.1	161.5
	LBT	145.7	145 4	155.3	154.6
	FCFS	150.0	151.3	161.5	157.7
I/O work time	RR	205.4	205.5	195.2	165.4
	STR	188 9	190.6	179.7	147.3
	SBT	195.1	196 9	184 9	148.9
	LBT	186.1	183.5	169 3	141.4
	FCFS	189.2	188 1	171.9	144.5
Total CP and I/O work time	RR	364.3	366.8	368.6	335.7
	STR	336 7	340 8	340.2	303.7
	SBT	347 6	352 7	352.0	310.4
	LBT	331.8	328 9	324 6	296.0
	FCFS	339 2	339.5	333.4	302.2
I/O overlap time	RR	167.8	171 3	180.8	145.5
	STR	133 5	138.4	141.0	104.2
	SBT	150.1	155 4	160 7	115.8
	LBT	122.7	122.1	123.6	92.1
	FCFS	135.4	136 3	136.1	102.0
Average flow time for the completed jobs	RR	11.8	11.6	15.4	58.8
	STR	11.7	11.8	14 8	54.3
	SBT	12.5	12.2	16.1	54.9
	LBT	12.5	13.2	17.1	57.0
	FCFS	13.2	13.5	19.2	55.7
Average wait time for the completed jobs	RR	7.8	7.8	10.8	51.0
	STR	7.8	7.9	10.0	46.7
	SBT	8.6	8.4	11.5	41.1
	LBT	8.7	9.6	12.7	49.2
	FCFS	9.4	9.7	14.4	47.8

lowed by SCF-RR. There are a number of striking and perhaps not intuitively obvious results from the simulation model.

There is an enormous difference in the throughput rates produced by the different job scheduling algorithms. Two methods, STF and SCF give strikingly better performance, over a factor of 2, over the SMF or FCFS method. Two facts of particular interest are that the SCF algorithm is so close to the STF algorithm in terms of throughput. Bearing in mind that the job mix was taken from well-grounded empirical measurements, it suggests that the SCF discipline with its more equitable selection of jobs is almost as good as the STF discipline with respect to throughput. The second feature is that the FCFS job scheduling discipline is so very poor. This suggests that queueing models which normally rely upon the use of first-come-first-serve scheduling disciplines only may predict erroneous throughput results for realistic job mixes. It is clear that with a job high arrival rate such as taken for this model, preemptive resume job scheduling will yield a higher throughput than non-preemptive scheduling. For the high throughput algorithms STF, and SCF, the improvement was in excess of 40 percent, a striking difference.

Central memory utilization as a performance measure would normally be of interest only in sharply memory limited systems. The only marked difference in the performance of any of the job scheduling algorithms is that the SMF produced a markedly lower central memory utilization. This would be expected since loading shortest memory first would tend to deplete the supply of small jobs which could be used to fill small gaps in the memory. This would lower the probability that a small residue of memory could be utilized effectively.

Multiprogramming increases CPU utilization and I/O channel utilization, and the mean degree of multiprogramming may be used as a measure of system performance. Note that there is a certain point in multiprogramming such that no performance improvement can be achieved even with a higher degree of multiprogramming. The SMF models with a higher degree of multiprogramming give a worse over all system performance than that of other models. Thus the degree of multiprogramming will not be used to evaluate system performance.

The measure of computer system performance most commonly used is the utilization of the central processor. The utilization of the CPU was very high for all memory (job) scheduling disciplines, and indeed for all CPU scheduling disciplines. This is indeed true for the computer system from which the experimental data used to characterize the model and the job mix were taken. The performance measures run typically about 5 percent greater than the actual performance of the system. This is due to omission of certain effects due primarily to queueing for peripheral processors and of certain aspects of system overhead. In the case of the exponential CPU service time distribution, the CPU scheduling algorithm had very little effect under any of the job scheduling disciplines. This is to be expected. On the other hand, in the case of the hyper-exponentially distributed CPU burst time, the CPU utilization varied more than 7 percent with different CPU scheduling algorithms. This span of CPU utilization is in good accord with the trace-driven model results of Sherman,

TABLE IV—Summary of Results for Hyperexponential CPU Burst Time Distribution Models

Throughput (Number of jobs completed)	STF-RR	STF-SBT	STF-STR	SCF-RR	SCF-STR	STF-FCFS	SCF-SBT
Degree of multi-programming	SMF-FCFS	SMF-LBT	SMF-SBT	SMF-RR	SMF-STR	SCF-FCFS	SCF-LBT
CM utilization	FCFS-FCFS	FCFS-LBT	FCFS-SBT	FCFS-RR	FCFS-STR	STF-FCFS	STF-LBT
CP utilization	SMF-RR	FCFS-RR	SMF-SBT	SMF-FCFS	SCF-RR	FCFS-SBT	SMF-STR
I/D Utilization	SCF-RR	STF-RR	SCF-SBT	SMF-RR	STF-SBT	SCF-STR	STF-FCFS
Mean flow time for the completed jobs	SCF-RR	STF-STR	SCF-STR	STF-RR	SCF-SBT	STF-LBT	STF-SBT
Mean wait time for the completed jobs	SCF-RR	STF-RR	STF-STR	SCF-STR	SCF-SBT	STF-SBT	STF-LBT
Overhead	FCFS-SBT	FCFS-STR	FCFS-LBT	FCFS-RR	FCFS-FCFS	SMF-FCFS	SMF-LBT

Baskett, and Browne.[9] It is interesting to note that for the hyper-exponentially distributed CPU service times the RR scheduling produced consistently the best results while on the exponentially distributed CPU burst time, the theoretically best[9] scheduling algorithm, SBT produced the best results consistently. This is particularly due to the neglect in the simulation model of the overhead in switching the processor from job to job. For non-preemptive job scheduling, the STF and SCF job scheduling gave *higher* CPU utilization than preemptive job scheduling. This is associated with the higher average degree of multi-programming for these cases with non-preemptive scheduling.

Other interesting results are obtained by considering total I/O utilization as a measure of performance. This measure is clearly affected by both job scheduling algorithms and CPU scheduling algorithms. The algorithm for utilization of I/O facilities was first-come-first-serve. In each, the order of I/O utilization is directly analogous to the throughput as a performance measure. It is particularly interesting to note that the RR central processor scheduling disciplines produced markedly higher utilization of I/O facilities over any other disciplines. As would be expected on theoretical grounds, SBT produced the next best results. Note that for the exponentially distributed CPU burst time case the SBT produced the higher utilization of I/O facilities. Preemptive job scheduling tended to produce higher rates of I/O utilization.

A measure well correlated with total I/O utilization is I/O overlap time. This is the total amount of time in the 180 seconds of the simulation runs that I/O processing and CPU processing were going on simultaneously. In some cases more than one I/O activity was overlapping a given CPU burst time.

Average wait time for completed jobs is an interesting measure of non-productive consumption of resources. During the wait time no processing, either I/O or CPU service was being applied to a given job. Thus, the larger this measure the more waste of central memory. It is interesting to note that in the exponential CPU distribution case the STR-CPU scheduling discipline scores well for all job scheduling disciplines while RR scores well in the hyper-exponential distribution case. It is also worth noting that the SMF scheduling algorithm performs better here than on any other measure.

The number of swaps of jobs in and out of memory is a convenient measure of overhead in central memory management. It is quite clear from this measure that the STF and SCF scheduling discipline incur a markedly higher overhead as the price paid for improvement in the throughput. Recall that these swaps are generated by preemption of batch jobs when jobs with a higher priority under the given job scheduling discipline arrive in the input queue. Note that we have ignored in all discussions the difference in cost of the overhead of the different memory scheduling algorithms. The use of ECS as a swapping medium in our model justifies this neglect. For swapping to disks or drum a serious overhead would be incurred.

We summarize briefly the most significant points of this research:

(1) Pre-emption is a key element for high throughput job scheduling.
(2) Job scheduling has a dramatic effect on throughput. It would appear that with our realistic job mix, the SCF is the most desirable job scheduling algorithm.
(3) If RR does not incur a high overhead for processor switching, it would appear to be the most desirable scheduling algorithm for CP scheduling if the CPU burst times have a strongly skewed distribution function (which is usually the case).
(4) Total I/O utilization is fairly strongly dependent on both memory scheduling algorithm and CP scheduling algorithm. For the case of a skewed distribution of CPU service times RR results again give a good utilization of I/O facilities.
(5) The scheduling disciplines which yield the highest throughput on the whole tend to incur the largest overhead.
(6) If maximum throughput or minimum mean flow time is the performance goal, then probably SCF memory scheduling and RR central processor scheduling (SCF-RR) or STF-RR will yield most consistently the best results.
(7) Either RR or a predictive scheduling mechanism based on attempting to predict that job which will have shortest burst time (SBT) will yield best CPU utilization.
(8) To maximize I/O utilization, SCF-RR or STF-RR would appear to be the most desirable combinations.

ACKNOWLEDGMENT

This research was supported by the National Science Foundation under grant GJ-1084.

REFERENCES

1 F BASKETT J C BROWNE W M RAIKE
The management of a multi-level non-paged memory system
Proc AFIPS 1970 SJCC Vol 36 AFIPS Press Montvale NJ pp 459-465

2 P R KLEINDORFER C H KRIEBEL
Analyzing job mix in multi-programmed computer systems

Management Sciences Research Report No 166
Carnegie-Mellon University August 1969

3 K D RYDER
A heuristic approach to task dispatching
IBM Systems Journal 8 3 1970 pp 189-198

4 W A WULF
Performance monitors for multi-programming systems
Proc 2nd Symposium on Operating Systems Principles
October 1969 pp 175-185

5 J E THORNTON
Design of a computer system: The Control Data 6600
Scott Foresman & Co Glenview Illinois 1970

6 Control Data Corporation
Control Data 64/65/6600 Computer Systems Reference
Manual Pub No 60100100 1967

7 H D SCHWETMAN
A study of resource utilization and performance evaluation of large-scale computer systems
TSN-12 Computation Center University of Texas Austin
Texas July 1970

8 J LAN
A simulation study of job and CPU scheduling
TSN-21 Computation Center and Computer Science
Department University of Texas Austin Texas December
1971

9 S SHERMAN J C BROWNE F BASKETT
Trace-driven modeling and analysis of CPU scheduling in a multi-programming system
To appear CACM—Also Proc of ACM Workshop on
Performance Evaluation Cambridge Mass pp 173-199
April 1971

Storage organization and management in TENEX

by DANIEL L. MURPHY

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

In early 1969, BBN began an effort aimed at developing a new time-shared operating system.* It was felt at the time that none of the commercially available systems could meet the needs of the research planned and in progress at BBN. The foremost requirement of the desired operating system was that it support a directly addressed process memory in which large list-processing computations could be performed. The cost of core storage prohibited the acquisition of sufficient memory for even one such process, and the problems of swapping such very large processes in a time-sharing environment made that solution technically infeasible as well.

Paging was therefore the logical alternative, and our study and experience with list processing systems[1,2] led us to believe that using a demand-paged virtual memory system for such computations was a feasible approach.

With demand paged process virtual memory added to our requirements, we found no existing system which could adequately meet our needs. Our approach was to take an existing system which was otherwise appropriate and add the necessary hardware to support paging. The system chosen was the DEC PDP-10,[3] which, although not paged, was available with a time-shared operating system and substantial support software.

Consideration was given to modifying the existing PDP-10 operating system to support demand paging, but that approach was rejected because of the substantial amount of work which would be required, because of the inherent constraints imbedded in the architecture of any large system, and because development of a new operating system would allow the inclusion of a great many other features and facilities which were judged desirable. Among these were a multiprocess job structure with software program interrupt capabilities, an interactive and well human-engineered command language, and advanced file handling capabilities.

Reports of some of the other operating system development in progress at the time suggested that considerable advantages were obtained by generalizing the concept of file storage and integrating process memory with it. Earlier systems had taken the view that files were sequential streams of bytes or words, perhaps with a facility for limited random accessing built on top.

In these earlier systems, process memory was viewed as the equivalent of the physical core memory that a program would see when running stand-alone on a dedicated processor. Time- and core-sharing facilities provided a means for several independent processes to use core and processor concurrently, but the basic concepts still required, for example, a file to be "read in" byte-by-byte or block-by-block into process memory.

The file-process memory integration achieved by MULTICS[4,5] provided an entirely different view of these concepts, and opened up many new possibilities for improved throughput, enhanced ease of programming, etc. The MULTICS segmentation concepts however, would have required substantial modification of the address computation logic of the processor and in other ways seemed to require a level of effort inappropriate to the scale of system we could support. Therefore, we began to examine the ways by which some of these same goals could be achieved in a system which had only paging hardware.

It was known from that outset that our system would contain multi-level storage components. A high speed, rapid access drum would obviously be needed as the swapping facility to support demand paging, and a larger and slower disk storage device (at least 50 million words) was planned for permanent storage. We were

* The work reported here was supported in part by the Advanced Research Projects Agency of the DOD, and in part by BBN.

already using a system, the XDS-940[6] which provided a means of "naming" process storage, and swapping on the basis of the named elements in a process memory. Although the file system was not integrated into this process memory naming scheme, certain basic concepts, e.g., a process memory map into which named elements could be placed, were present.

Thus, having determined that we would build a new monitor system to achieve certain specific objectives, we decided to adopt a more advanced architecture and obtain many other useful features. In particular, we realized that very little if any additional complexity was necessary in the design of the paging hardware in order to provide the base on which a monitor with integrated file and process memory could be built.

The system which resulted from this development effort is called TENEX, and this paper describes the facilities for naming memory and dealing with named memory which were developed and implemented in TENEX. Implementation details of the system are given, including the operation of the three levels of storage, and the flow of data between them.

NAMED MEMORY

TENEX terms and conventions

The discussion which follows will require knowledge of a few of the terms and conventions used in TENEX. The operating system provides a job structure which may contain multiple processes. By a job, we mean a set of active resources normally under control of a single user. That set may in principle be empty, but in practice will always contain at least one process.

In TENEX, each process is provided with an independent process address space, and is capable of performing computation in parallel with other processes. That is, TENEX processes are independent virtual machines with all necessary storage for holding the state of a computation. Various means are, of course, provided for allowing communication and control between processes.

File storage naming

The first and most obvious memory "name" in TENEX is the file name. A powerful and versatile directory and file naming facility is provided in which a particular file is identified by a fixed-depth path which includes device, directory name, file name, extension, and version.

The identifiers in each field (except for device and version) are strings of up to 39 characters. All permanent storage resides in files, so the first step in identifying any particular element of storage is to specify the path name.

It would be both cumbersome and inefficient to require that the file name be used for each operation on a file, even though TENEX provides default conventions which usually allow the user to specify only the name portion of the path. We therefore provide a means of associating the full path name with a small integer called a Job File Number (JFN) which will serve to identify the file over some limited period of time.

The JFN is an important concept in TENEX and deserves some further explanation. The first step in doing any operation on a file is to execute a monitor call giving as an argument the string representing the path name of the desired file.

Various conditions and default options are specified at that time. If the path name correctly identifies a single file, the monitor will return a JFN, and the association of that JFN with the file will remain in effect until the user program explicitly "releases" the JFN (or the job is logged out). JFN's are 18-bit numbers arbitrarily selected by the system, commonly but not necessarily assigned sequentially upward from 0. The domain of a JFN is the job in which it was assigned; therefore it may potentially be used by any process in the job (subject to various protection mechanisms). The system will always know what JFN's are in use in each job and so can assign at any time one known to be unique. It is possible for the same file to be associated with two or more JFN's within the same job (and with JFN's in other jobs), and this often happens when two processes are performing concurrent operations on the same file.

Once the initial association of JFN and file has been established, the JFN is used for all ensuing operations on the file, including sequential reading and writing, opening, closing, etc. The 18-bit JFN is a PDP-10 half-word, and so is conveniently manipulated by the system and user programs. Because the monitor system chooses JFN's to be indexes into system tables holding

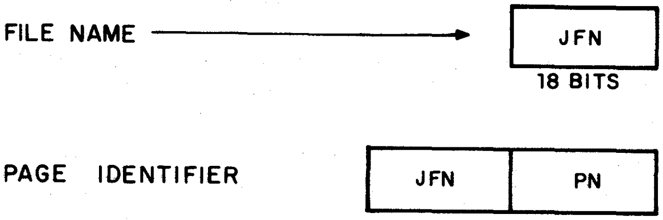


Figure 1

information about the relevant file, the lookup time on individual file function calls is very short and requires only a range test to reject invalid arguments.

Having once identified a particular file and obtained a JFN, a process need only identify the element within the file and the naming process will be complete. On a word-oriented machine such as the PDP-10, the most basic element in a file is obviously the word, but since we are operating in a paged environment, we will want to identify pages. Therefore, our complete identifier is constructed from the JFN of a file, and the page number (PN) within that file, as shown in Figure 1. The paging facilities will allow us then to reference any word within that page as described below.

File-to-process mapping

With the naming of our file memory specified, we next explain how this may be integrated with the address space of processes. As stated earlier, each TENEX process has an independent virtual memory of 256K words, a size fixed by the 18-bit addressing capability of the processor. With the TENEX page size of 512 words, each process virtual memory therefore consists of 512 pages. But these pages are not fixed storage. Rather, each page of the process virtual memory is actually a window through which one can look at a page of "real" storage.

To specify the contents (possibly null) of these windows, TENEX provides a virtual memory map, with one entry for each page of the virtual memory. Each map location is identified by a *map handle* which consists of two items, the process handle (provided by the system when the process was created), and the page number of the desired slot (Figure 2). It is important to understand that the map handle identifies a map slot and does not represent the contents (if any) of that slot. The monitor provides two basic operations for which the map handle is necessary, obtaining the identifier of the present contents of the slot, and placing an identified page into the slot.

This brings us to the basic facility for file/process memory integration. We have constructed a file system in which each page can be named with a convenient (one word) identifier, and we have specified a paged process address space represented by a map into which page identifiers can be put. Figure 3 shows this graphically. The process address space contains pages from two files, indicated by identifiers in the process map which act as pointers to the file pages.

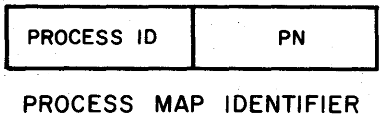


Figure 2

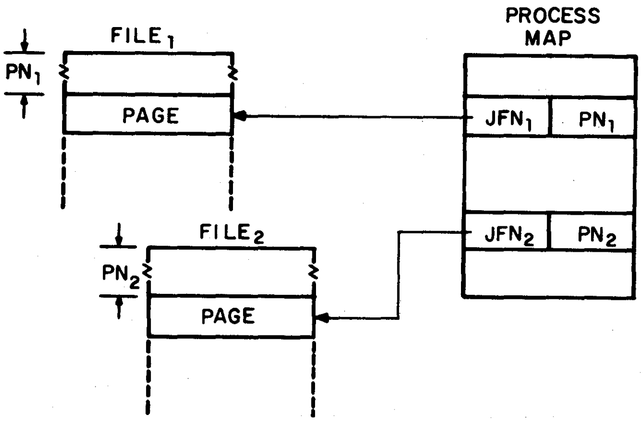


Figure 3—File-to-process mapping

There is some additional information in the map slots not included in these page identifiers, and that is the *access* permission. The TENEX paging hardware provides independent read, write, and execute access control on each page, so when a process places a file page identifier in its map, it must specify which of these accesses (each represented by a bit) is allowed. The system may further restrict the access according to arguments given when the file was opened, which in turn are limited to combinations permitted by the general protection mechanisms associated with file names. Thus the access actually permitted to a mapped page is the logical *AND* of the specific case access request (specified by the process) and the general access permitted to the file (specified by information residing in the file directory).

Sharing named storage

Since the file path names identify files over the domain of all jobs in the system, it is evident that our naming and mapping procedures readily provide a means for sharing storage. Using the appropriate path names (including legality checks), processes in two or more different jobs can identify the same file, and each can obtain a JFN for it. Nothing in the mapping procedures specified above requires that either process be aware of the other's access, and so each process constructs an identifier and places it in its process map (Figure 4). Remember that the JFN is associated with

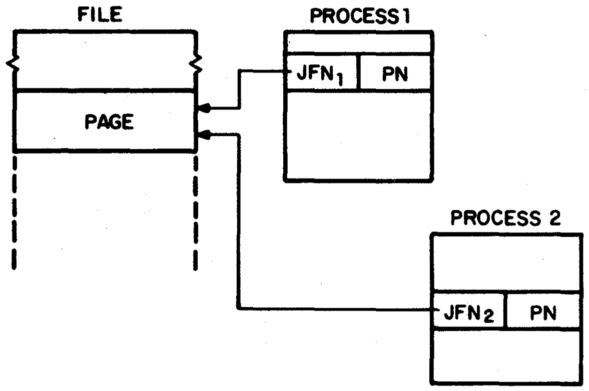


Figure 4—Shared file page

a file only within the domain of a job, and so the two JFN's shown are probably not the same small number. The page number (PN) shown is an absolute address within the file and will appear as the same number in both process maps. Thus two or more processes in the same or different jobs can identify and map the same page of physical storage. The mechanism by which this is implemented is described below.

Along with this basic sharing mechanism, TENEX provides a convention to help ensure that the access to shared or potentially shared information is logically consistent. We identify two cases:

1. A file contains information which must be in a consistent state to be used, e.g. a symbolic text file. Such a file may be read concurrently by several processes, but one process modifying the file precludes any other processes reading or modifying it.
2. A file contains information which, by agreement of the processes involved, can be simultaneously modified and used by several processes, e.g., a common data base or a file used for interprocess communication.

When a process opens a file, it must specify which of these two cases applies. The system will not permit any file to be open both ways at the same time on the grounds that such a situation can only result from disagreement among the processes on how the file is to be used, and is therefore a logical programming error. The monitor will permit any number of simultaneous case 2 openings of a file (which we call *thawed* access), and will allow any of the three types of access legal for the file to be used for each opening. The consistency and integrity of the data in the file is the responsibility of the processes using it.

The monitor will permit any number of case 1 openings of a file (which we call *frozen* access) providing all processes request only read and/or execute access. One or more openings of any type will preclude a new opening for write, and one write opening will preclude any new openings of any type. Thus the system guarantees the integrity of file data by prohibiting potentially conflicting access.

Copy-on-write access

One other important TENEX feature which facilitates sharing is a type of page access called copy-on-write. To our knowledge, this facility was first developed and used on the BBN-LISP system for the XDS-940[7]. It was developed as the result of two common observations:

1. Some programs, particularly older ones, are not quite reentrant. That is, they were coded without observing reentrant coding practices with the result that some code or initial data areas may be modified. Because of the architecture of the PDP-10, we in fact find many programs with completely reentrant code (even lazy programmers usually use the stack-oriented subroutine call and return instructions of the machine), but with local temporaries, data areas, etc., sprinkled arbitrarily through the program.
2. Some programs use large initial data bases which are common to all users, but which may be modified by some users in some specific cases. The principal example of this is the BBN-LISP system which initially contains over 100,000 words of compiled function code (reentrant), and some common list structure. It is however, necessary and legal for some users and some functions to modify portions of this base for local operations. In fact, none of this original base can be guaranteed immune from modification. For example, a list may be appended to, or a compiled function may have a "break point" temporarily inserted.

In TENEX, a process may specify this copy-on-write access whenever a file page is mapped into a process. Copy-on-write is legal even if write access is not. A page mapped in this way will remain shared so long as the process only does read or execute references. A write reference to the page will be trapped by the monitor, whereupon a private copy of the page will be made, and the process map changed so that it points to the copy rather than the original. Write access is then permitted to the copy, and the process' original write reference is completed.

All of this is invisible to the process, except that it may read its memory map and discover a different identifier and access than was initially used to map the page. This facility thus provides a means for allowing sharing wherever possible without penalizing unavoidable modifications or requiring the user program to handle them explicitly.

Examples of use of named memory

Let us consider the most common example of how file/process memory integration and sharing is used in TENEX, i.e., a file containing a commonly used program. We will identify this file as PROGRAM.SAV (the extension SAV by convention implies a core-image file). The file contains a number of pages of code and some mapping information as shown in Figure 5. The mapping information specifies where the code and data pages are to be placed in a process map to produce an image of the program. A monitor routine interprets the mapping information and performs the mapping. As shown in the figure, the code and data pages are arranged contiguously in the file, but may be put anywhere in the process map. In fact, the mapping shown is a common one, with data and temporary storage assigned to low addresses, and reentrant code assigned to addresses in the upper half of the process address space.

One might suggest that instead of placing pointers to the file in the process map, the file map itself be used as the process map. This would be analogous to running in a particular segment in the MULTICS-type segmentation scheme. But without the full power of general segment addressing, inter-segment references are not possible, and our procedure offers the following advantages.

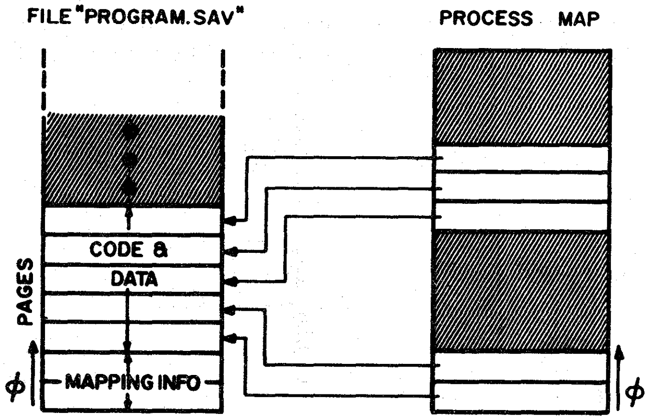


Figure 5

1. A process map may contain pages from several different files. In our scheme, individual pages or groups of pages may be viewed as mini-segments, and used in similar ways.
2. Different processes may have different access permissions to the same file page. In particular, when a write reference is done to a copy-on-write page, only one entry of the process map is changed to address the copy.

Sequential file access

While mapping operations are readily suggested in the case of program core images, it must be noted that the only basic type of file access permitted under TENEX is page mapping. TENEX provides a number of monitor facilities for other types of file access, the most common of which is sequential. To implement the file sequential monitor calls (e.g., byte-in, byte-out) the monitor maintains a number of "window" pages in a separate map invisible to the user process. For each file with sequential operations in progress, the monitor maps the file page which is to receive or provide the next byte. Each call from the user causes one or more bytes to be loaded from or stored into this page, and a count updated to determine if a new page should be mapped. Movement through the file is accomplished by mapping successive pages, and the sequential access module does not have to be aware of the physical device on which the page resides nor interface with I/O driver modules to read or write it. This modularity is very satisfying from an operating system design point of view.

As a final example, we note that processes may use shared file pages for interproccess communication. In this case, a particular file and set of pages within the file are agreed upon by several processes, and the pages are mapped into the address space of each of the processes. The actual map slots chosen by the processes need not be the same, i.e., the shared pages may be put in different places in the various process address spaces. Since the same physical storage is seen by all processes, any of a number of common techniques may be used to pass information in any direction, e.g., flags, ring buffers, etc.

In itself, this procedure does not provide any direct means for processes to signal one another, so for asynchronous events the processes are required to periodically test flag words in one of the shared pages.

IMPLEMENTATION

Pager

As stated above, paging hardware was designed and built as part of the TENEX development, and a few of

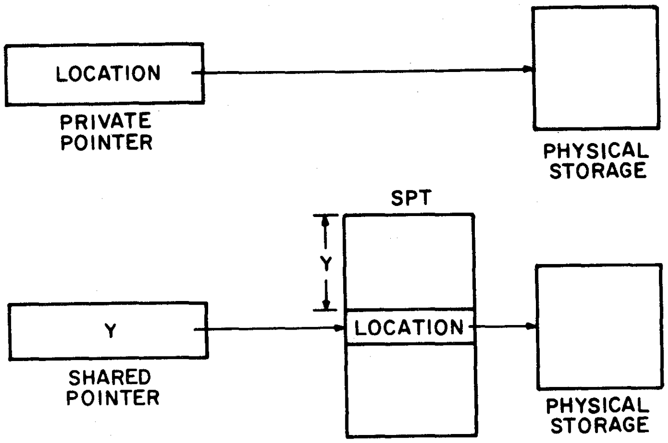


Figure 6—Pointer types

the characteristics of the BBN Pager are particularly relevant to this discussion. The pager is placed logically between the processor and the core memories and translates each memory address received from the processor into a physical core address which is sent to the memories. Control signals allow the pager to know what type of access the processor is making (read, write, or execute), and allow the pager to signal the processor when for some reason a reference cannot be completed (e.g., the page is not in core). The virtual addresses received from the processor are 18-bits, and the page size is 512 words, so the pager is in fact translating the high-order 9 bits of address, and passing the low-order 9 bits through unchanged.

The pager uses a set of associative registers to hold some number of recent virtual/physical address associations, but the source of this information is always a "page table" in core memory. Page tables contain (or point to) the physical storage address, if any, of each page of a virtual memory. Thus, each process virtual memory is represented by one page table. Page table entries are one word, hence a page table for a 256K virtual memory is 512 words, or exactly one page.

The pager references the relevant page table, using the 9 high-order virtual address bits as an index, whenever the associative registers fail to contain the requested virtual address. It is capable of interpreting three types of page table entries of which two are of interest here. The first is called a "private" pointer and contains a physical storage address. If this is a core address, the pager will load an associative register with the information and complete the requested reference. If it is any other address, the pager will initiate a trap to the monitor for appropriate action. The second type of page table entry is called a "shared" pointer, and contains an index into a system table at a fixed location. This "shared pages table" (SPT) contains the physical storage address, and the details of its function are described below.

These two pointer types are shown in Figure 6. The third type of page table entry is the "indirect" pointer described in Reference 8, but it is not relevant to this discussion.

One other fixed table, called the Core Status Table (CST), is used by the pager. For each page of physical core, this table contains information about recent references and notes if the page has been modified.

Hierarchical storage considerations

In any system using hierarchical storage, one is concerned with the movement of data between the various levels, with knowing where the current "up-to-date" copy is, with updating lower levels, etc. It is usually considered essential that the address of the currently valid copy of an item of storage reside in one and only one place. This tends to conflict with the goal of sharing which says that items of storage should be made available to many processes simultaneously. Replication of addresses would appear to admit the possibility of unresolvable phase errors, and the updating problem by itself would introduce undesirable complexity in the software.

One quite elegant solution to this problem is the hash table scheme which is shown in Figure 7. In this scheme, storage addresses reside in only one place, the storage hash table. Processes using an element of storage are given the "home" (and presumably invariant) address of the element, and the current location at any time may be found by performing a hash lookup into the table. Using this scheme, storage elements may be moved from place to place at any time, and only the table entry need be changed. Also, the table entry itself may be deleted when the element is moved back to its home address even though one or more processes are still using it. In this case the hash lookup will fail, and the monitor will have to re-create the entry.

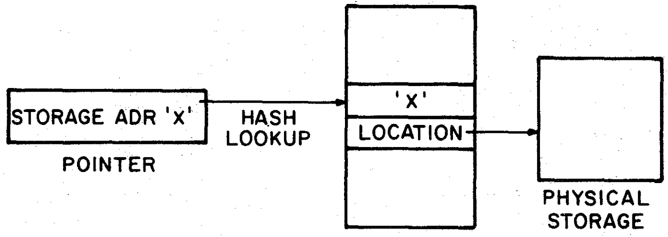


Figure 7—Hash table scheme

A second solution to the basic storage management problem is the shared pages table scheme used in TENEX and shown in Figure 6. In this scheme, storage addresses (for shared elements) again reside in only one place, a fixed table called the shared pages table. Processes using an element of storage are given a fixed index 'Y' which identifies the SPT entry holding the current address. Here, also, storage elements may be moved from place to place by changing only one address, but unlike the hash table scheme, an entry cannot be deleted from the SPT so long as pointers exist which use it. Therefore a share count is required for each entry to record the number of pointers to it which have been created.

We considered both of these schemes and a number of variations for TENEX before choosing the second of the above approaches. An exhaustive justification of this decision cannot be given here, but the decision was based primarily on our judgment that:

1. The cost of hardware to implement the hash table scheme was somewhat higher in terms of design effort and overall size and complexity.
2. Additional (time) overhead would be incurred in making the one or more probes into the hash table for each associative register reload.
3. The resident storage requirement of the hash table scheme would be greater.

TENEX implementation—mapping

We are now ready to show exactly how TENEX implements the file mapping operations discussed in the previous section, and how data flows between the several levels of storage. The TENEX storage hierarchy consists of three levels, core, swapping, and file. In practice, the swapping device is a fixed head drum with high transfer rate and fairly short latency time (e.g., less than 30 ms.). The file storage device is usually a movable head disk with substantially greater capacity, but reduced transfer and latency speeds.

As described in the previous section, named memory consists of pages within files, so we start with an example file and two of its pages as shown in Figure 8. The basic structure of the file is an index block containing the storage addresses of all of the data pages. This index block is in fact a page table, initially containing private pointers. We assume a starting point where none of the file pages are mapped in any process, so the "one and only one" place for the storage address of each of these file pages is logically and properly the index block of the file which owns them.

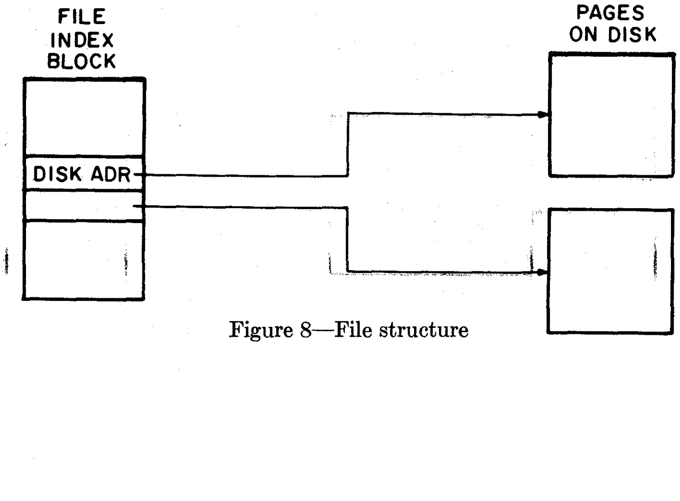


Figure 8—File structure

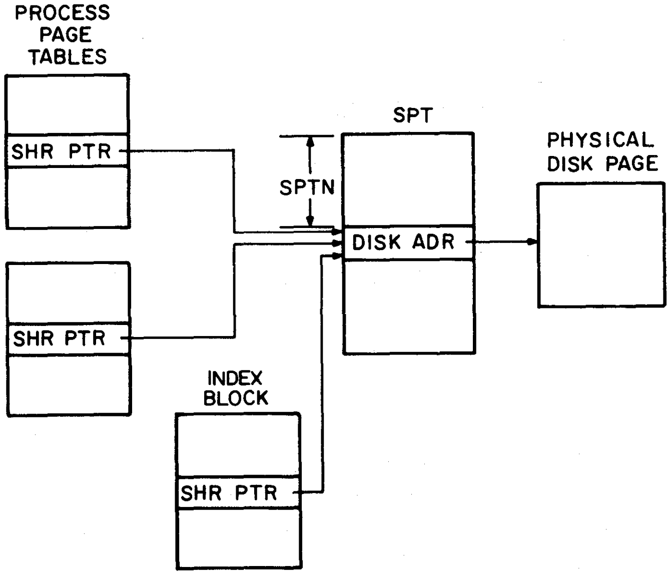


Figure 9—Two processes map a file page

Next, a process requests that one of these file pages be mapped into its address space. The monitor uses the JFN portion of the identifier to locate the file index block, and the PN (page number) portion to select the appropriate entry within it. Although our aim here is to have just one process using the page, we see that in fact the page must become shared at this point, that is, shared between the file and the process. Therefore, the monitor will assign a slot in the SPT and place in it the disk address obtained from the file index block. Simultaneously, it creates a shared pointer which points to that SPT slot and places a copy in both the file index block and the process page table. The share count for the SPT slot is set to reflect the fact that the page is in use twice, once by the file, and once by a process. A second process wishing to use the page proceeds in the same manner, but now it is only necessary to create another copy of the shared pointer and increment the share count. This situation is shown in Figure 9. The subsequent reduction of the share count to 1 (when all processes unmap the page) will indicate that the SPT entry may be reclaimed.

Some additional bookkeeping is necessary in order to keep track of the owner of the page, and the fact

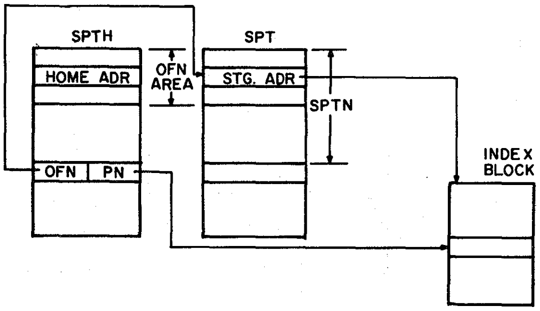


Figure 10—Ownership back pointers

that the file index block is in use. This is shown in Figure 10. The table labeled SPTH is a table parallel to and the same length as the SPT. For our example file page which was assigned slot 'SPTN', the parallel entry in the SPTH records the owning page table of the page. This is shown as OFN and PN. The OFN (open file number) is the monitor internal equivalent of the user's JFN, except that it identifies open files over the domain of all jobs in the system. The OFN is actually an index into a portion of the SPT which is reserved for index blocks, and the PN is the page number supplied by the user. The OFN portion of the SPTH holds the home addresses of the currently in use index blocks. The monitor must always open files on the basis of the storage address of the index block as obtained from the file directory, and a search of this part of the SPTH is necessary to determine if the file is already open.

Inter-level data flow

Next we show what happens when one of the processes references the file page which has been mapped. This is shown in Figure 11. The pager interprets the shared pointer found in the process map, and references the SPT. It finds, however, that the page is not in core and traps to the monitor. The monitor in turn selects a page of real core and initiates a read of the to disk bring in the page. The SPT slot is then changed to indicate that the page is in core.

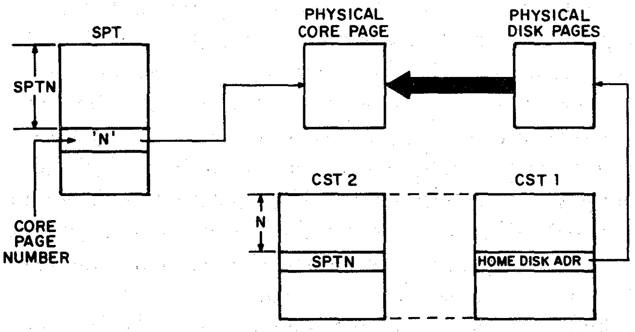


Figure 11—Page is referenced and brought into core

For completeness, we must note the function of two tables which record the state of physical core. These are the Core Status Tables (CST1 and CST2). For each page of physical core, CST1 holds the physical address of the next lower level of storage for the page. In our current example, this is a disk address because the page is just being read from the disk. CST2 records the name of the page table holding the pointer to that core page, which in this case is an SPT index. One additional bit (not shown) is used to record whether the page has been modified with respect to the next lower level of storage.

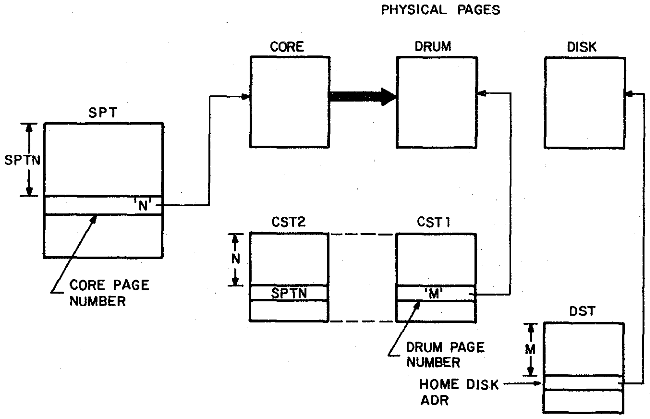


Figure 12—Page is swapped onto drum

Next we consider what is necessary for the monitor to swap the page onto the drum. It is important to note that during the course of the drum write (including latency) and for a period of time thereafter, the core page still contains a current copy of the data, and so we may properly leave the SPT slot pointing to it. This will prove useful in the event that a process makes another reference to the page during this time because the page will not have to be read into core again. Thus to begin the swapout, the monitor selects a free drum page, initiates the drum-write operation, and updates CST1 to reflect the fact that the next lower level of storage is now the drum.

However, we can't discard the home address of the page, so one other table is required. The DST (drum status table) serves a function for the swapping level of storage equivalent to that of the CST for core. That is, for each page in use on the drum, the DST holds the address of the next lower level of storage. It also records' whether the copy on the drum has been modified with respect to the copy on the disk so that the monitor will

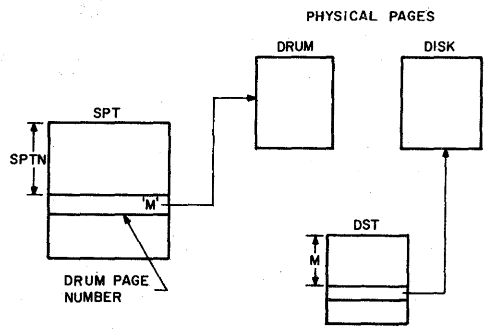


Figure 13—Core page is released

know whether a write is necessary at some time to update the disk copy. Our picture of a file page with copies on all levels of storage is now complete (Figure 12).

One final step is shown in Figure 13. If the page remains unreferenced for some period of time, the monitor will want to use the core page for some other purpose. To do this, the monitor will move the drum address from CST1 of the page being reclaimed to the SPT slot, and succeeding attempts to reference the page will discover that it is no longer in core.

Updating lower levels

So long as the page remains mapped by one or more processes, the share count will keep the SPT slot in use, and our convention is that the page will be moved between the drum and core as needed. This suggests that some procedure may be necessary to periodically update the home (disk) copy of pages. This is necessary both to guard against loss due to system crash, and because some files are mapped when the system starts up and are never unmapped (e.g., the disk assignment bit table). In TENEX, a special system process takes this responsibility. It periodically scans the open files, finding pages which have been changed since being read from the disk. File pages are backed up to the disk by setting a request bit in the CST which causes the swapper to move the page to the disk instead of the drum. File index blocks must also be updated but require a different procedure. For these, the backup process constructs an image of the index block as it would appear with no pages shared. That is, it finds the home address of each page and puts it in the index block in the form of a private pointer. This copy is then written on the disk. This procedure is a compromise of the goal or having only one copy of a storage address, but a simple interlock mechanism prevents any phase errors during the updating.

Dynamic storage management

One of the most important and difficult aspects of storage management in TENEX is the dynamic control of core and flow between levels of storage. The pager provides information on the frequency and type (read/write) of references made to pages in core. It also provides information on which of the processes sharing a page (i.e., having it mapped) have actually referenced it. A detailed description of these facilities and the algorithms which have been developed to handle dynamic storage management is beyond the scope of this paper.

SUMMARY AND CONCLUSIONS

This discussion has shown how named memory can be incorporated in an operating system having only paging facilities, and how some of the advantages of segmentation are thereby obtained. Although there are limitations to this approach, it does have the advantage of considerably less complex hardware and software. To date, we have not found a way to use mapping to provide dynamically linked library subroutines, one advantage which segmentation does provide. One possible solution may be to build a library of self-relocating subroutines and provide a convention for mapping them in a portion of the address space which the calling process is not using. Unfortunately, the PDP-10 processor does not provide a convenient facility for self-relocating code.

We have found that the process memory map is an extremely useful facility for a number of purposes. It is true that the 256K virtual memory eliminates the need for overlaying procedures in most programs, but where this technique is still required, it is easily implemented simply by remapping groups of pages.

The implementation of a three-level storage hierarchy used in TENEX has proved to be workable in over two years of actual operation. The software complexity required for the maintenance of the various tables is perhaps greater than would be required had we adopted the hash-table approach, but it has nonetheless been a manageable and programmable system.

ACKNOWLEDGMENTS

In addition to the author, T. R. Strollo, R. S. Tomlinson, J. D. Burchfiel, and E. R. Fiala actively participated in the design of this implementation strategy. R. S. Tomlinson and J. D. Burchfiel did the logic design and checkout of the Pager. Appreciation is also due in large measure to J. I. Elkind and D. G. Bobrow whose inspiration, leadership, and support made the TENEX project possible.

REFERENCES

1 D G BOBROW D L MURPHY
The structure of a LISP system using two-level storage
Communications of the ACM Vol 10 No 3 March 1967

2 ———
A note on the efficiency of a LISP computation in a paged machine
Communications of the ACM Vol 11 No 8 Aug 1968

3 DIGITAL EQUIPMENT CORP
PDP-10 reference handbook Dec 1971

4 V A VYSSOTSKY F J CORBATO R M GRAHAM
Structure of the MULTICS supervisor
Proceedings AFIPS 1965 FJCC Vol 27 Pt 1 Spartan Books New York

5 R C DALEY P G NEUMANN
A general purpose file system for secondary storage
Proceedings AFIPS 1965 FJCC Vol 27 Pt 1 Spartan Books New York

6 B LAMPSON et al
A user machine in a time sharing system
Proceedings IEEE 54 12 Dec 1966

7 D G BOBROW D L MURPHY W TEITELMAN
The BBN-LISP system reference manual
BBN April 1969 pp 3.8–3.9

8 D G BOBROW J D BURCHFIEL D L MURPHY R S TOMLINSON
TENEX, a paged time sharing system for the PDP-10
Communications of the ACM Vol 15 No 3 March 1972

The application of program-proving techniques to the verification of synchronization processes

by KARL N. LEVITT

Stanford Research Institute
Menlo Park, California

INTRODUCTION

The purpose of this paper is to establish the applicability of program-proving techniques to the verification of operating systems, control programs and synchronization programs. All the illustrative examples to be presented use Dijkstra's[1] P and V operations for controlling the synchronization of competing processes. However, the techniques discussed are applicable to any set of such control primitives. A major portion of the paper is devoted to the proof of correctness of two programs devised by Courtois et al.[2] that control the sequencing of "readers" and "writers" requesting the use of a common device.

The notion of establishing the correctness of computer programs by providing a formal proof of correctness originated with Floyd[3] and Naur.[4] In the following section we discuss the Floyd-Naur approach in more detail, but for our purposes here the method can be summarized as follows. Each input line of a program is associated with an *input assertion* φ that expresses any constraints on the input variables. Similarly, each output line is associated with an output assertion ψ that expresses the desired relation among output variables when (and if) the program halts. Certain intermediate program lines, most notably those lines that serve to cut the program loops, are associated with *"Floyd"* assertions (or simply, assertions) that express the relationship among all program variables whenever control passes to those points.

The correctness of the program with respect to these programmer-supplied assertions is proved as follows. For each path in the program that commences and terminates with assertions (and not traversing any intermediate assertions), it is shown that the "commencing" or antecedent assertion, together with the transformation expressed by the intervening code, implies the "terminating" or consequent assertion. Such a proof establishes the correctness of the program, if successfully carried out for all paths, provided the program halts.* The establishment of program halting, as described by Floyd, is to be carried out as a separate proof, by, for example, showing a well-ordering of a variable's values throughout the execution of the program. Manna[5] later showed that a single proof process, albeit more difficult than Floyd's, and also undecidable, could suffice to demonstrate both program correctness and halting. London[6,7,8] has proven some moderate size programs (up to 100 lines of code) using Floyd's method, and has provided some insight into the specification of the assertions. The status of program proving through June 1971 is summarized in a tutorial manner in a survey paper by Elspas et al.[9]

In most applications of program proving to date, it has been assumed that (1) the program is executed in a *serial,* i.e., uniprocessing, uniprogramming environment, or that any multiprogramming or multiprocessing is invisible to the program's execution; and (2) the program contains specified output points to which well-defined output assertions can be applied. The implication of assumption (1) is that, at any instant of time, control resides at only one point in the program. In contrast, synchronization and operating system programs are parallel programs. There are really two types of parallelism of concern to us. *Explicit* parallelism occurs when more than one processor is available to execute a program so that several program paths are processed simultaneously. *Implicit* parallelism, which is

* We emphasize that this proof process establishes program correctness with respect to the user-supplied assertions. As another way of looking at the process, the proof establishes the equivalence between a procedural description (i.e., the program) and a nonprocedural description (i.e., the assertions).

really multiprogramming occurs when a program's execution can be temporarily interrupted to carry out the execution of another program. Thus several programs are in various stages of execution at any instant.

With regard to assumption (2) the behavior of an operating system is not conveniently described by assertions placed at output points. In particular an operating system does not contain output points since t should, in proper operation, never halt.

The dominant theme here relates to techniques for handling the simultaneous or parallel activity associated with an operating system and for the specification and proving of Floyd assertions so that the intent of the program is distributed among these assertions.

Some mention should be made of previous work relating to the proving of programs with parallel activity. Ashcroft and Manna[10] have investigated a particular model of a parallel program wherein several independent parallel paths may exist in a program (and it is conceived that each path has its own program counter), but execution is carried out by a uniprocessor. In executing the program the uniprocessor arbitrarily selects one of the paths, processes the single instruction specified by the program counter, and then arbitrarily selects the next path to consider, which might also be the previously considered path. BEGIN- and JOIN-type nodes are also included for generating and "collapsing" parallel paths. Ashcroft and Manna then visualize a single nondeterministic program, based on the original parallel program, that contains *choice points* corresponding to the several instructions from which the uniprocessor may select at any instant. Assertions are applied to each such choice point and to all points at the heads of loops. The assertions at a particular point describe the state of all program variables when (and if) control reaches the point in question, for all possible paths of control to that point. Thus the problem of proving a parallel program reduces to the proving of a conventional program provided suitable interpretation is given to the choice point.

A disadvantage of this approach is that a proliferation of assertions results from the need to consider every instruction in each path as a choice point. To alleviate this situation Ashcroft and Manna introduced a special block into the model so that, if control enters this block, the execution of the block is continued without interruption until the block halts.

The parallel model that we will consider here for the synchronization programs is an extension of the Ashcroft-Manna model in that processing is carried on simultaneously by more than one processor. In addition, we assume the existence of a special SPLIT node that permits a uniprocessor path to be converted into a multiprocessor path and a *critical section* so that only one processor at a time is granted access to such a section. With these improvements the number of assertions tends to remain manageable.

In a recent paper Habermann[11] has made an initial attempt to formalize the synchronization mechanisms associated with control primitives like P and V, and has provided proofs of several simple programs using such primitives. In a sense, Habermann's proof technique could eventually be more attractive than the method we will describe since it takes better advantage of the hierarchical or modular structure induced by the control primitives. The present disadvantage of the Habermann method is that it is ad hoc for programs with a mixture of control primitives and "conventional" code. As the program-proving field matures so that automatic or semiautomatic program verifiers become available, our approach will be amenable to implementation by such a system. In fact, the proofs that emerge although lengthy, involve relatively simple manipulations, and should be implementable by relatively unsophisticated program verifiers.

In the following sections we review briefly the pertinent program-proving theory, review the semantics of the P and V operators and present a proof of a simple mutual exclusion program, present a few simple extensions to handle the parallel case, present detailed proofs of the two programs by Courtois, and present our conclusions.

REVIEW OF PERTINENT PROGRAM-PROVING THEORY

In the foregoing section we pointed out that in proving a program by Floyd's method the user must provide an input assertion φ for each input point, and an output assertion ψ for each output point. In addition, he must provide intermediate assertions $q_1, q_2, \ldots$, so that each loop in the program is cut by at least one such q_i. The process of proving the program with respect to the applied assertions is to prove each path, where a path is defined by an antecedent assertion, a consequent assertion, and intervening code. For each path a *verification condition* is derived that is a statement of the form:

$$\text{antecedent assertion} \wedge \text{intervening code} \supset \text{consequent assertion.}$$

In order to prove the program correct it is necessary and sufficient to prove that each of the verification conditions is logically correct. An interesting mechanical approach to the generation of verification conditions called *back substitution* has been developed by King.[12] Since we will later make use of back substitution in developing verification conditions, it is worth giving it a brief discussion here for program code consisting of simple variable assignment statements of the form $y \leftarrow$(expression) and branch statements. For the treatment of array assignment statements of the form $A(m) \leftarrow$expression and of procedure calls, the reader is referred to References 12, 13, and 14.

Briefly, the generation of verification conditions for a path with antecedent assertion q_i and consequent assertion q_j involves the carrying out of string substitutions proceeding backward along the path from q_j to q_i. That is, if q_j involves a variable y and a simple assignment statement immediately preceding q_j is of the form $y \leftarrow f(x)$, then q_j is transformed to a q_j', where each occurrence of y in q_j is replaced by $f(x)$. The newly formed assertion q_j' is then transformed to a q_j'' based on the assignment preceding $y \leftarrow f(x)$ and so on, until an assertion q_j^n is generated so that all assignments between q_i and q_j are accounted for. The verification condition to be proved to establish the correctness of the path in question is $q_i \supset q_j^n$. A test (or branch statement) T, appearing subsequent to the generation of an assertion q_j^k, is handled by transforming q_j^k to $T \supset q_j^k$. Substitutions specified by subsequent assignment statements (proceeding backwards) are made for variables in both q_j^k and T.

The mechanics of back substitution are illustrated with respect to the simple program path depicted below. The program is taken from Reference 9.

```
      $q_1$
Test: $P-A-B \ge 0$
      $Y \leftarrow Y+D/2$
      $A \leftarrow A+B$
      $B \leftarrow B/2$
      $D \leftarrow D/2$
      $q_1$
```

The guessed assertion is given by q_1: $[A=(Q*Y)] \wedge (B=Q*D/2) \wedge (D=2^{-k}) \wedge (k=\text{nonnegative integer}) \wedge (P/Q-D<Y \le P/Q)$. The reader can verify that the back substitution of q_1 through the intervening tests and assignment statements leads to the following verification condition:*

$$q_1 \wedge (D/2 \le E) \wedge (P-A-B \ge 0) \supset (A+B=Q*(Y+D/2) \wedge (B/2=Q*D/4) \wedge (D/2=2^{-k}) \wedge (k=\text{nonnegative integer}) \wedge (P/Q-D/2<Y+D/2 \le P/Q).$$

We leave it to the reader to verify that the above condition is logically true. For example, we note that the term $(B=Q*D/2)$ in q_1 implies the term $(B/2=Q*D/4)$ in the consequent of the verification condition.

At present, we know of several implementation of verification condition generators that handle the simple assignment statements and tests discussed above, in addition to arrays, procedure calls, and various ALGOL-like constructs. The discussion section contains a prognosis of the availability of verification condition generators, theorem provers, and semiautomatic assertion generators.

P AND V PRIMITIVES AND PROOF OF THE SIMPLE MUTUAL EXCLUSION PROBLEM

Dijkstra[1] introduced the P and V operators as a software approach toward controlling the access to critical sections of competing processes.* The simplest possible use of these operators is illustrated by the following program,

↓
P(S)
↓
critical section
↓
V(S)
↓

wherein it is assumed that many processes wish to gain access to the critical section, but only one such process is to be processing the critical section at any instant. The P operator ensures that a process gains access only if no other processes have current access; otherwise, the requesting process is forced to wait. The V operator activates the scheduling of a deferred process on the completion of the current processing of the critical section. (Throughout the discussion we will assume that the scheduler arbitrarily selects one of the deferred processes for access to the critical section.) The parallelism here is actually trivial; control can be in the critical section and P simultaneously, but at no other pair of points simultaneously. The following interpretation of P and V will accomplish the desired control**

* We have made use of the tautology $[R \supset (S \supset T)] \equiv [(R \wedge S) \supset T)]$ to form a logical expression involving a single implication where only the transformed consequent assertion appears on the right side of the implication.

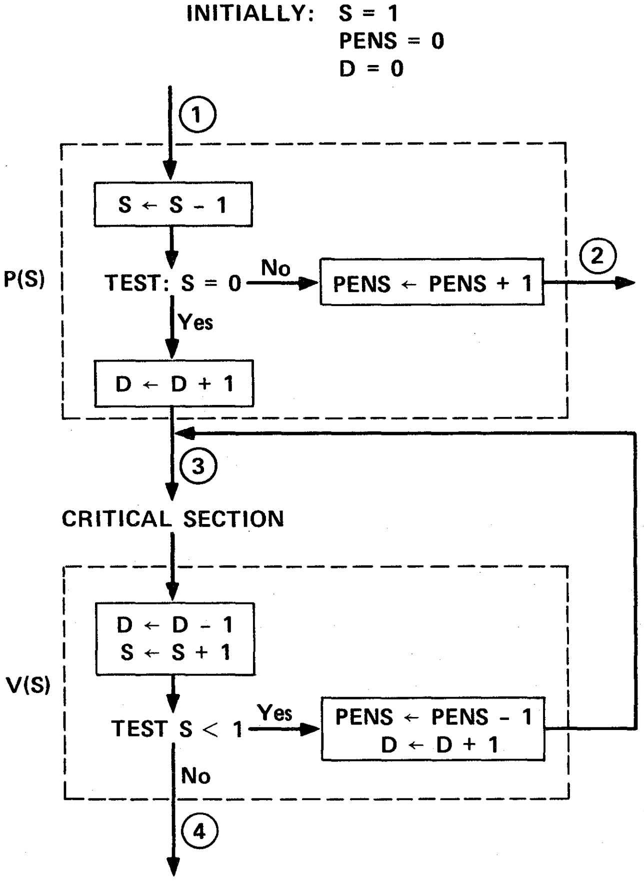
* Throughout this discussion we will assume a *process* to mean a task that requires access to particular resource or resources for its execution.

** We assume the existence of some primitive lock-out mechanism so that only one process at a time gains control of a *P* or *V* operator.

wherein, according to Dijkstra, the variable S serves as a *semaphore.*

```
Initially  S=1,
P(S):      S←S−1;
           if S=0 then schedule process else wait.
V(S):      S←S+1;
           if S<1 then schedule deferred process else
           done.
```

To prove formally that the control is indeed as hypothesized, we will represent the program as the flow chart of Figure 1. We have introduced two new integer variables: PENS, which indicates the number of processes pending (on the semaphore S), and D, which indicates the number (hopefully 0 or 1) of processes that are processing the critical section. (Note that PENS and D are not strictly a part of the P and V mechanism but merely variables that we have introduced to simplify the extraction of the program's intent from the assertion.) Point ② of the flow-chart corresponds to the *wait* point, and when the V operation schedules a process, control returns to Point ③. Points ① and ④ correspond, respectively, to the entry point of a process and the exit point wherein no processes are pending on S.



TA-710582-33

Figure 1—Flow chart representation of simple control program

The proving of the program has two aspects. The first part, which we will call the *correctness* part, is to prove that at any instant D is either 0 or 1, corresponding to 0 or 1 processes in control of the critical section. The second part, which we will call the *deadlock* part, is to show that $D \geq 1$ if and only if $PENS \geq 1$. The proof of this latter condition will ensure that, if a set of processes has been deferred, then there is a process that will eventually perform a V operation and schedule a deferred process. This approach toward avoiding deadlock has been called the *expediency* condition,[17] and ensures that the system never reaches a state where no requests can be granted. The deadlock part of the proof corresponds to the proof of halting in Floyd's method in that in both cases the proof is handled apart from the proof of correctness.

The use of program-proving techniques requires the attachment of assertions to the flow chart. We have assumed that if a process has gained control of the P (or V) operators, then all other processes are prevented (by hardware lockout) from gaining control of either P or V until the process in control has taken either of the two exits from P (or V). On the basis of th's assumption it is not necessary to apply assertions at any interior points in the P or V operations, since the state of the variables at the terminal points of P and V are sufficient to specify the state at any interior points of P or V. However, by the definition of the problem this does not apply to the critical section, i.e., if the control were not working as intended then several processes could have control of the critical section.

The "guessed" Floyd assertions for the program are as follows, where q_i, $i=1, 2, 3, 4$ is the assertion at point (i).

$$\begin{aligned}
q_1 &= (\text{integer S}) \wedge (S \leq 1) \wedge (D = u(-S+1)) \\
&\quad \wedge (PENS = u(S) - S) \\
q_2 &= (\text{integer S}) \wedge (S < 0) \wedge (D = 1) \wedge (PENS = -S) \\
q_3 &= (\text{integer S}) \wedge (S \leq 0) \wedge (D = 1) \wedge (PENS = -S) \\
q_4 &= (\text{integer S}) \wedge (S = 1) \wedge (D = 0) \wedge (PENS = 0),
\end{aligned}$$

where $u(x)$ is the step function defined by

$$\begin{aligned}
u(x) &= 0 \quad \text{for} \quad x \leq 0 \\
u(x) &= 1 \quad \text{for} \quad x > 0.
\end{aligned}$$

Two steps must be followed in proving the program with respect to the above four assertions. Step 1 is to prove that for all paths the assertions are consistent with the transformation specified by the intervening code; Step 2 is to establish that the validity of the correctness and deadlock parts is correctly embedded in the guessed assertions.

First, in Step 1 the following control paths must be verified:

$$1 \rightarrow 2,\ 1 \rightarrow 3,\ 3 \rightarrow 4,\ 3 \rightarrow 3.$$

For purposes of illustration we will outline the proof of 1→3; this outline should enable the reader to verify the other paths. The path from 1→3 embodies the following steps

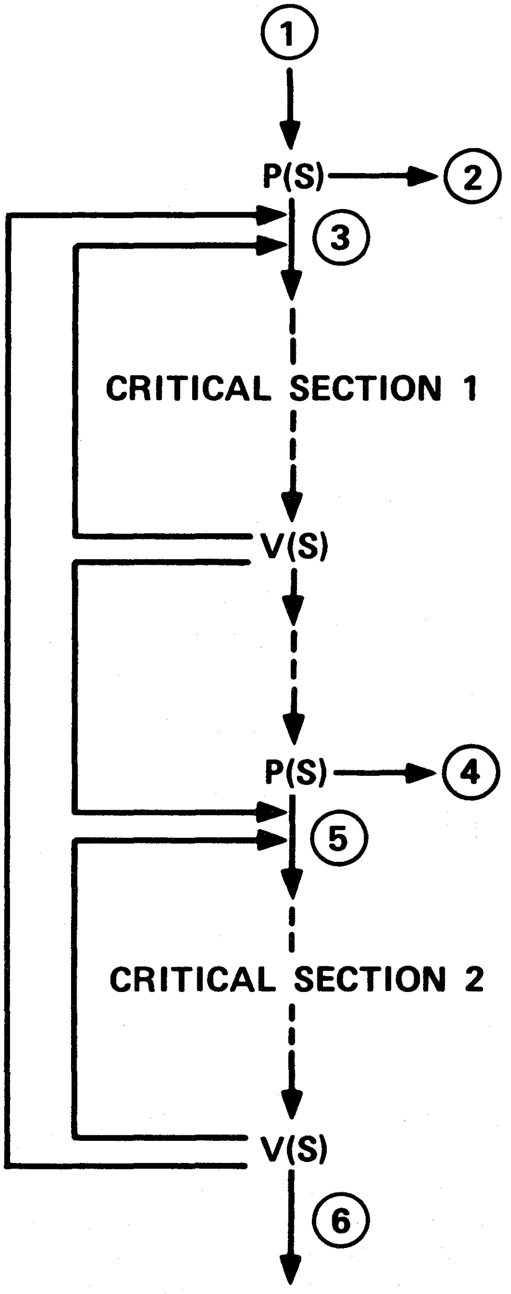
$$\begin{array}{c} q_1 \\ \mathrm{S} \leftarrow \mathrm{S}-1 \\ \text{Test:}\ \ \mathrm{S}=0 \\ \mathrm{D} \leftarrow \mathrm{D}+1 \\ q_3. \end{array}$$

Back substitution on q_3 leads to the following verification condition:

$$\begin{aligned} &[(\text{integer S}) \wedge (\mathrm{S} \leq 1) \wedge (\mathrm{D}=u(-\mathrm{S}+1)) \\ &\qquad \wedge (\mathrm{PENS}=u(\mathrm{S})-\mathrm{S})] \wedge (\mathrm{S}-1=0) \\ &\qquad\quad \supset [(\text{integer S}-1) \wedge (\mathrm{S}-1 \leq 0) \\ &\qquad\qquad \wedge (\mathrm{D}+1=1) \wedge (\mathrm{PENS}=-\mathrm{S}+1)]. \end{aligned}$$

The first term of the consequent, integer S−1, is true from integer S. The second term is true from $\mathrm{S}-1=0$, i.e., $\mathrm{S}=1$. The third term, $\mathrm{D}=0$, is true from $[\mathrm{D}=u(-\mathrm{S}+1)] \wedge (\mathrm{S}=1)$. The fourth term is true from $[\mathrm{PENS}=u(\mathrm{S})-\mathrm{S}] \wedge (\mathrm{S}=1)$. Thus the path 1→3 is verified (with respect to the "guessed" assertions).

Step 2, establishing that the assertions embody the desired behavior of the correctness and deadlock parts, remains to be carried out. The correctness part is apparent from the assertions by noting that $\mathrm{D}=0$ or 1. The deadlock part is satisfied by noting that whenever $\mathrm{PENS} \geq 1$, then also $\mathrm{D} \geq 1$; thus there exists a process currently in the critical section that will eventually schedule some deferred process.



TA-710582-34

Figure 2—A control program with two critical sections

As an extension of this simple control program that we will use in the following sections, consider the program displayed in Figure 2. The program is a straightforward extension of the simple single critical section program discussed above. It can be shown by a proof similar to that outlined above that access is granted to only one of the two critical sections at a time. Thus, control cannot be simultaneously at points ③ and ⑤. The interpretation of P(S) and V(S) is modified from that described previously, as shown in Figure 3. The variables PENS1 and PENS2 serve to indicate, respectively, the number of processes pending on semaphore S at critical sections 1 and 2. The "CHOOSE" box functions as follows. Either of the two output branches is chosen at random. If the test in the selected branch succeeds, then control continues along the branch; otherwise, control is passed to the other branch. Note that the relation $\mathrm{S}<1$ ensures that control

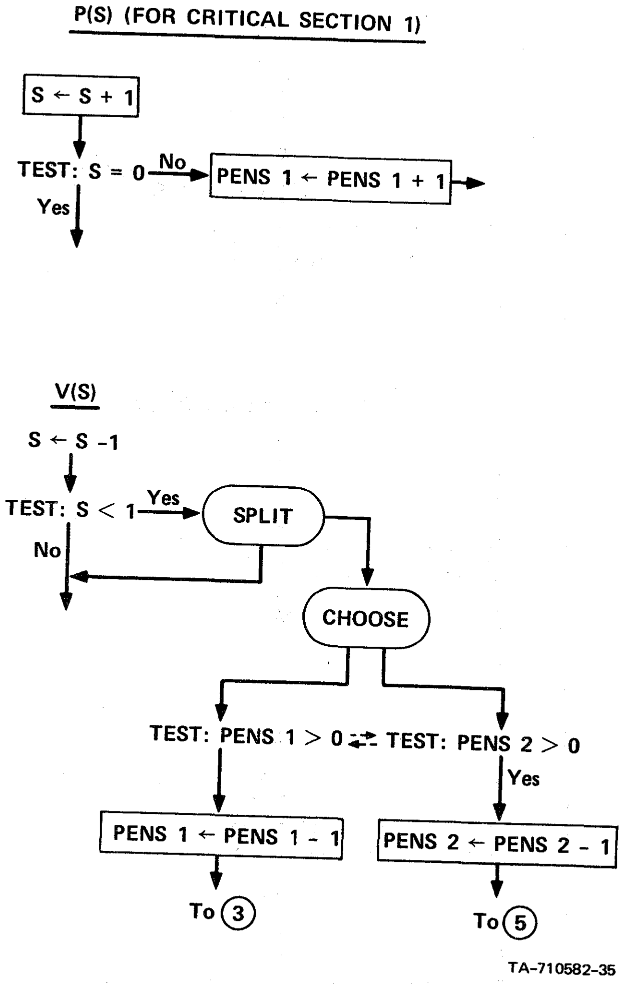


Figure 3—Interpretation of P and V for two critical sections

can pass along at least one of the branches because if S<1, then PENS1+PENS2>1. The purpose of the CHOOSE box is to place no arbitrary constraints on the scheduling of deferred processes. The "SPLIT" box simultaneously passes control along each of its output branches. The intention here is both to reschedule another process onto a critical section associated with semaphore S and to have the process that just finished the critical section execute the instructions following V(S).

Wherever two or more parallel paths converge there is a JOIN box, embodying some rules for combining the paths. Points ③ and ⑤ of Figure 3 are really JOIN boxes. The most apparent such rules are OR (AND) indicating that control is to proceed beyond the JOIN box wherever any (all) of the inputs to the JOIN box are active. Our discussion will apply mainly to the OR rule, but is easily extended to the AND case.

APPLYING ASSERTIONS TO SYNCHRONIZATION PROGRAMS AND ABSTRACTING THE PROOF OF CORRECTNESS AND DEADLOCK FOR THE ASSERTIONS

The simple program of Figure 1 reveals, although only in a trivial manner, the possibilities for parallel activity that we wish to exhibit. For example, in Figure 1 it is possible for control to reside simultaneously in the critical section (point ③) and at point ①. The assertion we applied at point ① reflects the possibilities for multiple points of control in that the variable relationships correspond to control being only at point ①, simultaneous at points ① and ③, or simultaneous at points ①, ②, ③. (It is assumed that processors are available to execute any code associated with the critical section as well as with the P(S) and V(S) blocks.) In proving the program we did not require any new formalisms beyond those associated with the uniprocessing situation since hardware locks are so constituted that the P and V operations are not simultaneously executed.

A more general situation is displayed in Figure 4. Here we illustrate portions of two processes, A and B, with interprocess communication achieved via the semaphore S. The particular model of computation that we will assume is as follows:

> Assume that at periodic intervals calls are made on sections A or B. The availability of a processor

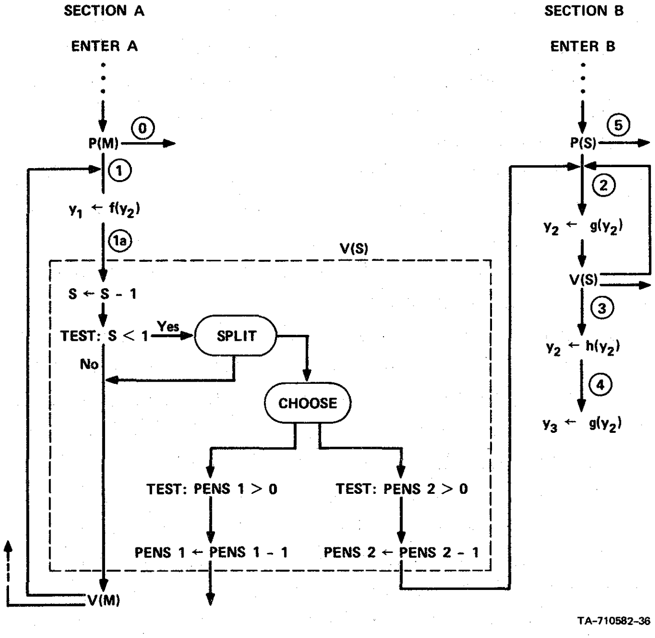


Figure 4—Program to illustrate assertion interpretation

to commence process'ng of the calls is always assumed to exist. If two or more processors attempt simultaneous reference to a variable or operator, the selection of the processor that achieves access is made arbitrarily. If execution is deferred, say, at point ⓪, subsequent to the P(M) operation, the affected processor is presumably freed to handle other tasks. When the corresponding V(M) operation is carried out, schedul ng a deferred process, a processor is assumed to exist to effect the processing.

With reference to this program and the assumed model of parallel computation, we will illustrate approaches to the placement of assertions and to proving the consistency of the assertions relative to intervening program statements.

Assertion placement

Since we are assuming a parallel/multiprocessing environment, there are potentially many points in the flow chart at which a processor can be in control. For example, in Figure 4 control can be simultaneous at points ①, ②, and ③. However, we will assume that the role of the P(M) and V(M) operations is to exclude simultaneous accesses to the intervening critical section, provided there are no branches into the critical section. Hence, control cannot be simultaneous at points ① and $\textcircled{1a}$. An assertion, for example at point ①, must reflect the state of the variables of the various processes assuming that:

(1) Control is at point ① and, simultaneously,
(2) Control is at any possible set of *allowable* points.

By "allowable" we mean sets of points not excluded from control by virtue of mutual exclusion. We recall that for the uniprocessor environment assertions are placed so that each path in the program is provable. As an extension of that observation we can show that the proving of paths in a parallel program can be accomplished provided the following rules are satisfied:

(1) Each loop in the program must be broken by at least one assertion.
(2) Within a critical section (i.e., one where control is at only one point at a time and where any variables in the critical section common to other portions of the program are themselves in critical sections under control of the same semaphore), only a sufficient number of assertions need be applied to satisfy the loop-cutting rule, (1). We assume that all entries to critical section are controlled by P, V primitives. If not then rule (3) below applies.
(3) All points not covered by rule (2) must generally be supplied with assertions.
(4) Each HALT point and all WAIT points associated with a P operation must contain assertions.

Thus, in Figure 5 a possible placement of assertions is at points ⓪, ①, ②, ③, ④, and ⑤. Note that since the purpose of synchronization programs is generally to exclude, by software techniques, more than one process from critical sections, such programs will not require the plethora of assertions associated with a general parallel program. Also note that it is a simple syntactic check to determine if a given assertion placement satisfies the above rules.

Once the points where the assertions are to be placed have been selected and the assertions have been developed, it remains to prove the consistency of assertions. As in the uniprocessor case, the first step in this proof process is to develop the verification conditions for each path. For the parallel environment of concern to us here, we are confronted with the following types of paths: simple paths, paths with SPLIT nodes, paths with CHOOSE nodes, and impossible paths. These four path types are handled below, wherein the rules are given for developing the verification conditions, and some indication is given that the parallel program is correct if these rules are followed. A complete proof of the validity of the rules is not given because an induction argument similar to that of Floyd's applies here.

Verification condition for a simple path

By a *simple path* we mean a path bounded by an antecedent and a consequent assertion, with the intervening program steps being combinations of simple branch and assignment statements. For such a path the verification condition is derived exactly as in the uniprocessor case. That this is the correct rule is seen by noting that the assertion q_a placed at point a in the program reflects the status of the variables, assuming that control is at point a and also at any allowable combination of other points containing assertions. Also note that because of our assertion placement rules, the variables involved in the code between a and b are not modified simultaneously by any other process. Thus, if a simple path $a \rightarrow b$ is bounded by assertions q_a and q_b and if it is proven that $q_a \wedge$ (intervening code) $\supset q_b$, then the path is proven independently of the existence of control at other allowable points.

Verification conditions for paths with SPLIT nodes

Assume that a SPLIT node occurs in a path, say, bounded on one end by the antecedent assertion q_a. Recall that at the SPLIT node, separate processors commence simultaneously on each of the emerging paths. Also assume that along the two separate paths emerging from the split nodes the next assertions encountered are q_b and q_c, respectively.* In this case the "path" (which is actually two paths) is proved by showing that

$$q_a \wedge (\text{code between point } a \text{ and SPLIT node}) \wedge (\text{code between SPLIT node and point } b) \wedge (\text{code between SPLIT node and point } c) \supset (q_b \wedge q_c).$$

Note that it is not sufficient merely to consider the path between, say, a and b, since the transformations between the SPLIT node and c may influence the assertion q_b. However, note that the variable references along the two paths emerging from the SPLIT node are disjoint, by virtue of the rules for selecting assertion points. Hence the use of back substitution to generate the verification condition can function as follows. Assertion q_b is back-transformed by the statements between point b and the SPLIT node, followed by the statements between point c and the SPLIT node, finally followed by the statements between the SPLIT node and point a to generate q_b. A similar rule holds for traversing backward from q_c to generate q_c. Note that the order in which the two paths following the SPLIT node are considered is not crucial since these paths are assumed not to reference the same variables.

Verification condition for a path with a CHOOSE node

Recall that when control reaches a CHOOSE node having two exits, the exit that is chosen to follow is chosen arbitrarily. Hence the effect of a CHOOSE node is simply to introduce two separate *simple* paths to be proven. For antecedent assertions q_b, q_c, what must be proved is

$$q_a \wedge (\text{code between } a \text{ and } b) \supset q_b$$
$$q_a \wedge (\text{code between } a \text{ and } c) \supset q_c.$$

Note that one or possibly both of the paths might not be control paths, but this introduces no difficulties, as we show below.

Impossible paths

As mentioned above, not all topological paths in a program are necessarily paths of control. In effect, what this means is that no input data combinations exist so that a particular exit of a Test is taken. Recall that for antecedent and consequent assertions q_a, q_b and an intervening Test, T, the verification condition is $q_a \wedge T' \supset q_b'$, where the prime indicates that back substitution has been carried out. Clearly, if the test always evaluates to FALSE, then $q_a \wedge T'$ must evaluate to FALSE, in which case the implication evaluates to TRUE independent of q_b'. (We recall that TRUE$\supset$TRUE, FALSE$\supset$TRUE, and FALSE$\supset$FALSE are all TRUE.)

Proving that program has no deadlock

For the parallel programs that we are dealing with deadlock will be avoided if for every semaphore S such that one or more processes are pending on S, there exists a process that will eventually perform a V(S) operation and thus schedule one of the deferred processes. (We are not implying that every deferred process will be scheduled, since no assumptions are made on the scheduling mechanism.) In particular, if a process is pending on semaphore a, then it is necessary to show that another process is processing a. If that latter process is also pending on a semaphore b, it is necessary to show that $b \neq a$, and that a third process is processing b. If that third process is pending on c, it is necessary to show that $c \neq b$, $c \neq a$, and that a fourth process is processing c, etc.

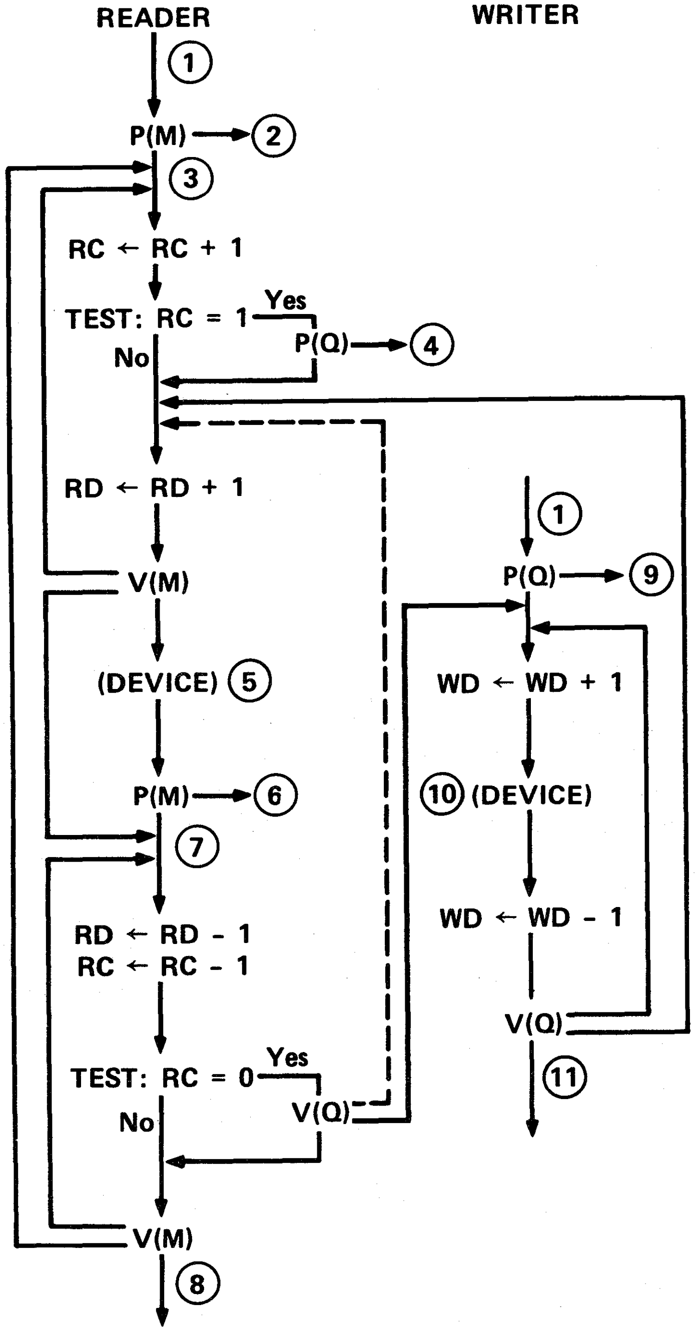
In the next sections we apply the concepts above to the verification of particular control programs.

PROOF OF COURTOIS[2] PROBLEM 1

This section presents a proof of a control program that was proposed by Courtois et al. The program is as follows:

```
Integer
  RC; initial value=0
Semaphore
M, Q; initial values=1
READER                       WRITER
P(M)
RC←RC+1
if RC=1 then P(Q)
V(M)                         P(Q)
  READ PERFORMED             WRITE PERFORMED
P(M)                         V(Q)
RC←RC−1
if RC=0 then V(Q)
V(M)
```

* Various special cases are noted, none of which introduce any particular difficulties. It is possible that q_a, q_b and q_c might not be all distinct or that another SPLIT node occurs along a path before encountering a consequent assertion.



TA-710582-37

Figure 5—Flow chart representation of Courtois problem 1

The purpose of the program is to control the access of "readers" and "writers" to a device, where the device serves in effect as a critical section being competed for by readers and writers. If no writers are in control of the critical section, then all readers who so desire are to be granted access to the device. (We show below that the program almost satisfies this goal, although under certain rare circumstances a reader's access might be deferred for a writer even though at the time at which the reader activates the READER section no writer is actually on the device.) A writer is to be granted access only if no other writer or reader is using the device; otherwise, the requesting writer's access is deferred. In particular, any number of simultaneous readers are allowed access provided no writers are already on. The role of the semaphore M is to enforce a scheduling discipline among the readers' access to RC and Q. For our model of parallel computation, it can be shown that the semaphore M is not needed, although its inclusion simplifies the assertion specification.

Figure 5 is a flow chart representation of the program. A few words of explanation about the figure are in order. The V(Q) operator for the reader and the V(M) operator for the upper critical section are assumed to be the generalized V's containing the CHOOSE and SPLIT nodes as discussed in the two previous sections. The other V operators are assumed to contain CHOOSE but no SPLIT nodes. The dashed line emerging from V(Q) indicates a control path that will later be shown to be an impossible path.

Associating appropriate variables with each of the P and V operators, the following integer variables and initial values are seen to apply to the flow-chart.

M	Q	RC	RD	WD	RPENQ
1	1	0	0	0	0

WPENQ	RPENM1	RPENM2
0	0	0 ,

where the R and W prefixes to a variable correspond, respectively, to readers and writers and the 1 and 2 suffixes correspond, respectively, to the "upper" and "lower" critical sections associated with semaphore M.

Once again we will divide the proof for this program into a *correctness part* and a *deadlock part*. For the correctness part we will establish that

(1) WD=0 or 1, indicating that at most one writer at a time is granted access to the device.

(2) If WD=1, then RD=0, indicating that if one writer is on the device, then no readers are "on."

(3) If WD=0, then RPENQ=0, indicating that if no writer is on the device, then a reader is not held up by semaphore Q. An entering reader under these circumstances could be held up by semaphore M, i.e., RPENM1>0. (We will temporarily defer discussion of this situation.)

According to the assertion placement rules, each input, output and wait point must possess an assertion, each loop must be cut by an assertion, and in addition, an assertion must be placed at each point along a path wherein along another parallel path there exists an instruction referencing variables common to the point in question. For this program the assertion placement problem is simplified since all variables, e.g., RC and Q, common to two or more parallel paths are a part of critical sections wherein access is granted only to one such critical section at a time. Hence, only the input-output and loop-cutting constraints must be satisfied, leading to a possible placement of assertions at the numbered points in Figure 5. Note that point ⑤ does not require an assertion, but since it represents a control point where readers are on the device, it is an interesting reference point.

The assertions associated with all 11 control points are indicated in Table I. The assertion labelled GLOBAL is intended to conjoin with the other 11 assertions. The appearance of (i) at the beginning of a disjunctive term in q_2, q_3, q_8, q_9 indicates that the first (i) terms are the same as in q_1. Thus, for example, in the first disjunctive term of assertion q_2, the first six conjunctive terms are the same as in the first disjunctive term of q_1, but the seventh and eighth terms are different, as shown.

It is worthwhile discussing our technique for specifying the assertions—we will provide sample proofs later on to attest to the validity of the assertions. In specifying the assertion at a point a, we assumed, of course, that control is at a and then attempted to guess at which other points control could reside. Variable relationships based on this case analysis were then derived, and then the expressions were logically simplified to diminish the proliferation of terms that resulted. For example, in assertion q_1, the first disjunctive term corresponds to the case: no writers on the device, i.e., control is *not* at ⑩. The second disjunctive term corresponds to the case of control at ⑩. With regard to the second term if control is hypothesized at ⑩, it is also guessed that control could possibly be at ⑨, ②, and ③ or ④.

It remains to verify all the paths bounded by the 11 assertions. The paths so defined are:

1→2; 1→3; 3→4; 3→(5, 3); 3→(5, 7); 5→6; 5→7, 7→8 [RC≠0]; 7→7 [RC≠0]; 7→3 [RC≠0]; 7→(5, 3) [RC=0]; 7→(5, 7) [RC=0]; 7→(5, 8) [RC=0]; 7→(10, 3) [RC=0]; 7→(10, 7) [RC=0]; 7→(10, 8) [RC=0]; 1→9; 1→10; 10→11; 10→10; 10→5; 10→(5, 3); 10→(5, 7).

A brief discussion of the symbolism is in order. For example, the path 3→(5, 3) commences at ③, and then splits at the SPLIT node of V(M) into two paths leading to ⑤ and ③. The path 7→(10, 3) [RC=0] indicates that the branch defined by RC=0 is taken, followed by a splitting at V(Q), one path leading to ③, and the other path taking the CHOOSE exit toward ⑩. Clearly, many of the above paths are impossible paths—as revealed by the proof.

We will not burden the reader of this paper with proofs of all the paths, but we will provide an indication of the proofs for several of the more interesting paths.

TABLE I—Assertions for Courtois Problem 1

Global:	$(\text{All variables} \in I)\wedge(M\leq 1)\wedge(Q\leq 1)\wedge(RC\geq 0)\wedge(RD\geq 0)\wedge(WD\geq 0)\wedge(RPENQ\geq 0)\wedge(WPENQ\geq 0)\wedge(RPENM1\geq 0)\wedge(RPENM2\geq 0)$
q_1:	$[(WD=0)\wedge(RD=RC)\wedge(RPENQ=0)\wedge(WPENQ=u(Q)-Q)\wedge u(Q)=u(1-RC))\wedge(RPENM2\leq RD)\wedge(RPENM1+RPENM2=u(M)-M)]\vee[WD=1)\wedge(RD=0)\wedge(RPENQ=RC)\wedge(WPENQ=-Q-RC)\wedge(RC=u(RC))\wedge(RPENM2=0)\wedge(RPENM1=u(M)-M)\wedge(M\leq u(1-RC))]$
q_2:	$[(6)(RPENM1>0)\wedge(M<0)]\vee[(7)M<0]$
q_3:	$[(7)(M\leq 0)]\vee[(7)(M\leq 0)]$
q_4:	$[(WD=1)\wedge(RD=0)\wedge(RPENQ=1)\wedge(WPENQ=-Q-1)\wedge(RC=1)\wedge(RPENM2=0)\wedge(RPENM1=-M)\wedge(M\leq 0)\wedge(Q\leq 0)]$
q_5:	$[(WD=0)\wedge(RD=RC)\wedge(RPENQ=0)\wedge(WPENQ=-Q)\wedge(Q\leq 0)\wedge(RPENM2\leq RD-1)\wedge(RPENM1+RPENM2=u(M)-M)\wedge(RC\geq 1)]$
q_6:	$[(WD=0)\wedge(RD=RC)\wedge(RPENQ=0)\wedge(WPENQ=-Q)\wedge(Q\leq 0)\wedge(RPENM2\leq RD)\wedge(RPENM2\leq 1)\wedge(RPENM1+RPENM2=-M)\wedge(M<0)\wedge(RC\geq 1)]$
q_7:	$[(WD=0)\wedge(RD=RC)\wedge(RPENQ=0)\wedge(WPENQ=-Q)\wedge(Q\leq 0)\wedge(RPENM2\leq RD-1)\wedge(RPENM1+RPENM2=-M)\wedge(M\leq 0)\wedge(RC\geq 1)]$
q_8:	$[(5)(RPENM2=0)\wedge(RPENM1=0)\wedge(M=1)]\vee[WD=1)\wedge(RD=0)\wedge(RPENQ=0)\wedge(WPENQ=-Q)\wedge(RC=0)\wedge(RPENM2=0)\wedge(RPENM1=0)\wedge(M=1)]$
q_9:	$[(7)(Q<0)]\vee[(8)Q<0]$
q_{10}:	[Second disjunctive term of 1]
q_{11}:	$[WD=0)\wedge(RD=0)\wedge(RC=0)\wedge((RPENQ=0)\wedge(WPEN=0)\wedge(Q=1)\wedge(RPENM2=0)\wedge(RPENM1=u(M)-M)]$

TABLE II—Proof of Path 10→(5,3) in Courtois Problem 1

```
                  Program steps:
                        q10
                  WD←WD−1
                  Q←Q+1
            Test: Q<1
            Test: REPENQ>0
              RPENQ←RPENQ−1
              RD←RD+1
              M←M+1
            Test: M<5
                  ↙      ↘
   Test: RPENM1>0          q5
      RPENM1←RPENM1−1
                 q3
```

Backsubstitute q_3 and q_5 to yield q_3', q_5'

q_5': $(WD=1)\wedge(RD+1=RC)\wedge(RPENQ=1)\wedge(WPENQ=-Q-1)\wedge(Q+1\leq 0)\wedge(RPENM2\leq RD)\wedge(RPENM1+RPENM2$
$=u(M+1)-M)\wedge(RC\geq 1)$

q_3': $[(WD=1)\wedge(RD+1=RC)\wedge(RPENQ=1)\wedge(WPENQ=u(Q+1)-Q-1)(u(Q+1)=u(1-RC))\wedge(RPENM2\leq RD+1)$
$\wedge(RPENM1+RPENM2=u((M+1)-M)]\vee[(WD=2)\wedge(RD=-1)\ldots]$

Tests backsubstituted

T': $(RPENM1>0)\wedge(M<0)\wedge(RPENQ>0)\wedge(Q<0)$

Verification Conditions

$q_{10}\wedge T'\supset q_5'$ $\quad q_{10}\wedge T'\supset q_3'$

Sample Proof: Proof of q_5' term RPENQ = 1

From q_{10}: $(RPENQ=RC)\wedge(RC=u(RC))$

Thus RPENQ = 0 or 1

From T′ RPENQ > 0

Thus RPENQ = 1

Table II outlines the steps in proving the path 10→(5, 3). At the top of Table II we delineate the steps encountered along the path. As is readily noted, the path contains a SPLIT node. To develop the verification condition, back substitution is required from both q_3 and q_5 to form q_3' and q_5'; note that in developing q_5' the statements between the SPLIT node and point ③ must be considered, in addition to the statements directly between points ⑩ and ⑤. To verify the path, the following two logical formulas must be proved true: $q_{10}\wedge T'\supset q_5'$, $q_{10}\wedge T'\supset q_3'$. At the bottom of Table II we outline the few simple steps required to prove the term (RPENQ = 1) in q_5'. The patient reader of this paper can carry out the comparably simple steps to handle the remaining terms. Note that q_3' is the disjunction of two terms, one beginning with the term (WD = 1) and the other with the term (WD = 2). For $q_{10}\wedge T'\supset q_3'$ to be true, it is necessary for only one of the disjunctive terms to be true. The reader can verify that it is indeed the first disjunctive term that is pertinent.

As a final note on the verification of paths, consider the path 10→(5, 7). A little thought should indicate that this should be an impossible path since the effect of control passing to point ⑦ is to schedule a process that was deferred at point ⑥, but at point ⑥ a reader is considered to be on the device, and hence point ⑥ could not be reached from point ⑩ where a writer is on the device. This is borne out by considering the formula $(q_{10}\wedge T')$ for the path in question. In q_{10} there is the conjunctive term (RPENM2 = 0) while in T′, the back-substituted test expression, there is the conjunctive term (RPENM2 < 0). Thus, $q_{10}\wedge T'$ evaluates to FALSE, indicating that the path is impossible.

It remains now to prove the correctness and deadlock conditions by observation of the assertions and the program itself. The key assertion here is q_1 since it expresses the relationship among variables for all possible variations of control, e.g., for all allowable assignments of processors to control points in the program. On the basis of q_1 we can conclude the following with regard to correctness:

(1) WD = 0 or 1, indicating that no more than one writer is ever granted access to the device.

(2) If WD = 1, then RD = 0, indicating that if a writer is on the device, then no reader is.

(3) The issue of a requesting reader not encountering any (appreciable) delay in getting access to a device not occupied by a writer is more complicated. From the first disjunctive term of q_1

(that deals with the case of no writers on the device), we note that if WD=0, then RPENQ=0. Hence, under the assumed circumstances a requesting reader is not deferred by semaphore Q. However, a requesting reader could be deferred by semaphore M. In fact, a requesting reader could be deferred at point ② while the RD readers on the device emerge from point ⑤, and then be scheduled onto the lower critical section wherein the last emerging reader performs V(Q) and schedules a writer. The deferred reader will then be scheduled onto the upper critical section only to be deferred by Q at point ④. Although it is an unusual timing of requests and reader completions that leads to this situation, it still violates the hypothesis that a reader is granted access provided no writer is on the device.* Note that, under the assumption that (WD=0) and RPENM2 remains zero while a requesting reader is deferred by M at point ②, the requesting reader will be granted access to the device prior to any requesting writers.

We now dispose of the question of deadlock. We need to demonstrate that, if a process is pending on a semaphore, then there exists another process that will eventually perform a V operation on that semaphore. With regard to semaphore Q, we note from observation of q_1 that if RPENQ>0 or WPENQ>0, then either WD=1 or RD≥1. Thus, if any process is pending on Q, there exist processes that might eventually do a V(Q) operation. It remains to dispose of the issue of these processes themselves pending on semaphores. It is obvious that a writer on the device must emerge eventually, at which time it will do a V(Q) operation. For one reader (or more) on the device, in which case RPENQ=0, we have shown that the last reader will perform a V(Q) operation. A reader could be deferred by semaphore M, but in this case there is a reader processing M that is not deferred by Q and hence must do a V(M) operation.

DISCUSSION

In this paper we have developed an approach based on program-proving techniques for verifying parallel control programs involving P and V type operations. The proof method requires user-supplied assertions similar to Floyd's method. We have given a method for developing the verification conditions, and for abstracting the proof of correctness and nondeadlock from the assertions.

We applied the technique to two control programs devised by Courtois et al. At first glance it might appear that the method is only useful for toy programs since our proofs for the above two programs seem quite complex. However, in reality the proofs presented here are not complex, but just lengthy when written out in detail. The deductions needed to prove the verification conditions are quite trivial, and well within the capability of proposed program proving systems.* By way of extrapolation it seems reasonable for an interactive program verifier to handle hundreds of verification conditions of comparable complexity. Thus one might expect that operating systems containing up to 1000 lines of high-level code should be handled by the proposed program verifier.

We might add that some additional theoretical work is called for relative to parallel programs and operating systems. Perhaps the main deficiency of the proofs presented here is that a suspicious reader might not believe the proofs. In establishing the correctness of the programs it was required to carry out a nontrivial analysis of the assertions. For example, we refer the reader to the previous section where the subject of a reader not encountering any delay in access is discussed. Contrast this with a program that prints prime numbers, wherein the output assertion says that the nth item printed is the nth prime—if the proof process establishes the validity of the output assertion there is no doubt that the program is correct. It is thus clear that the operating system environment could benefit from a specification language that would provide a mathematical description of the behavior of an operating system. Also some additional work is needed in understanding the impact of structured programming on the proof of operating systems. We would expect that structured programming techniques would reduce the number of assertion points and the number of paths that must be verified.

* We conjecture that there is no solution to this problem without permitting the conditional testing of semaphores, so that the granting of access to a writer or reader to the device is decided on the basis of the arrival time of a reader or writer at the entry point to the control program. In effect, what the program here accomplishes is to grant a reader access to the device provided it passes the test: RC = 0 while WD = 0. Note that there are other problems that do not admit to solutions using only P and V operations unless conditional testing of semaphores is permitted, e.g., see Patil.[15]

* See London[16] for a review of current and proposed program proving systems.

ACKNOWLEDGMENTS

The author wishes to thank Ralph London for many stimulating discussions on program proving and operating systems and for providing a copy of his proof of the two programs discussed in this paper. Peter Neumann, Bernard Elspas and Jack Goldberg read a preliminary version of the manuscript. Two referees provide some extremely perceptive comments.

REFERENCES

1 E W DIJKSTRA
The structure of THE multiprogramming system
Comm ACM 11 5 pp 341-346 May 1968

2 P J COURTOIS F HEYMANS D L PARNASS
Concurrent control with "READERS" and WRITERS"
Comm ACM 14 10 pp 667-668 October 1971

3 R W FLOYD
Assigning meanings to programs
In Mathematical Aspects of Computer Science
J T Schwartz (ed) Vol 19 Am Math Soc pp 19-32
Providence Rhode Island 1967

4 P NAUR
Proof of algorithms by general snapshots
BIT 6 4 pp 310-316 1966

5 Z MANNA
The correctness of programs
J Computer and System Sciences 3 2 pp 119-127 May 1969

6 R L LONDON
Computer programs can be proved correct
In Proc 4th Systems Symposium—Formal Systems and Nonnumerical Problem Solving by Computer R B Banerji and M D Mesarovic (eds) pp 281-302 Springer Verlag New York 1970

7 R L LONDON
Proof of algorithms, a new kind of certification (Certification of Algorithm 245, TREESORT 3)
Comm ACM 13 6 pp 371-373 June 1970

8 R L LONDON
Correctness of two compilers for a LISP subset
Stanford Artificial Intelligence Project AIM-151 Stanford California October 1971

9 B ELSPAS K N LEVITT R J WALDINGER
A WAKSMAN
An assessment of techniques for proving program correctness
ACM Computing Surveys 4 2 pp 97-147 June 1972

10 E ASHCROFT Z MANNA
Formalization of properties of parallel programs
Stanford Artificial Intelligence Project AIM-110
Stanford California February 1970

11 A N HABERMANN
Synchronization of communicating processes
Comm ACM 15 3 pp 177-184 March 1970

12 J C KING
A program verifier
PhD Thesis Carnegie-Mellon University
Pittsburgh Pennsylvania 1969

13 D I GOOD
Toward a man-machine system for proving program correctness
PhD Thesis University of Wisconsin Madison Wisconsin 1970

14 B ELSPAS M W GREEN K N LEVITT
R J WALDINGER
Research in interactive program-proving technique
Stanford Research Institute Menlo Park California May 1972

15 S PATIL
Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes
MIT Project MAC Cambridge Massachusetts February 1971

16 R L LONDON
The current status of proving programs correct
Proc 1972 ACM Conference pp 39-46 August 1972

17 R C HOLT
Comments on the prevention of system deadlocks
Comm ACM 14 1 pp 36-38 January 1971

APPENDIX

Proof of Courtois problem 2

Figure 6 displays the flow chart of the second control problem of Courtois et al.[2] The intent of this program is (1) similar to problem 1 in that the device can be shared by one or more readers, but a writer is to be granted exclusive access; (2) if no writers are on the device or waiting for access, a requesting reader is to be granted immediate access to the device; and (3) if one or more writers are deferred, then a writer is to be granted access before any reader that might subsequently request access. As we show below, a formal statement of the priorities can be expressed in terms of the variables of Figure 6. Also, as in problem 1, the intent of the program is not quite achieved relative to the receiving of requests at the entry points of the reader and writer sections.

It is noted that the program contains semaphores L, M, N, Q, S, all of which have initial value 1, and "visible" integer variables RS, RD, RC, WS, WD, WC, all of which have initial value 0. In addition, as in problem 1, there are the variables associated with the various P and V operations. As in problem 1, the V operators, with the exception of V(L) and those at points ⑩ and ⑱, embody both the SPLIT and CHOOSE nodes; V(L) has only the SPLIT node, and the final V's have only CHOOSE nodes. The dotted control lines indicate paths that can be shown to be impossible.

The numbered points on the flow chart are suitable for assertion placement in that each loop is cut by at least one assertion and all commonly referenced variables are contained within critical sections. There are several approaches toward deriving the assertions, but once again the most sensible one involves case analysis.

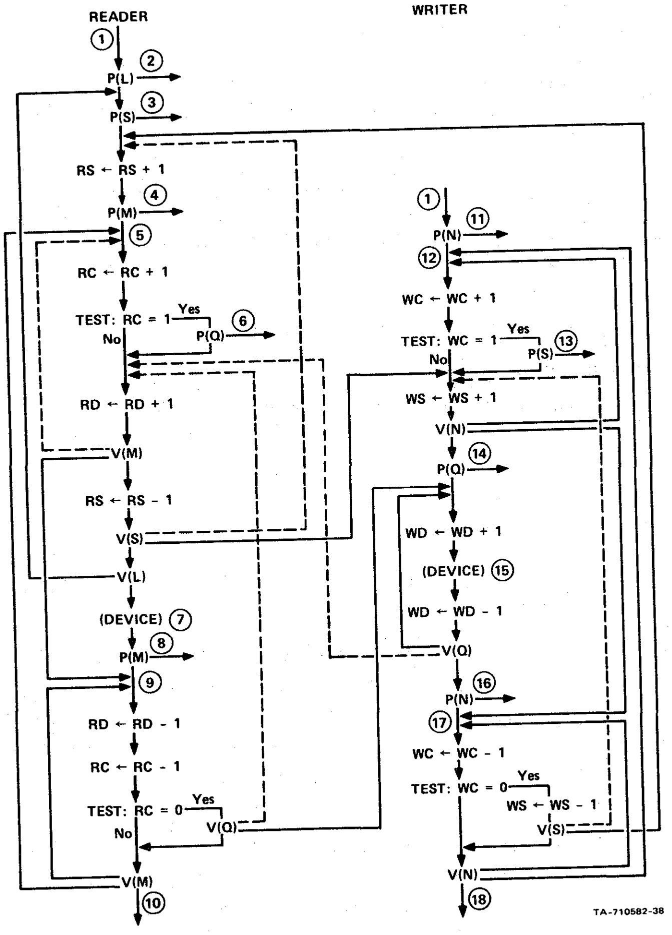


Figure 6—Flow chart representation of Courtois problem 2

From the view of control at point ①, we have derived the assertion q_1 of the form $q_1 = c_1 \wedge (a_1 \vee [a_2 \wedge (b_1 \vee b_2)])$, wherein a_1 corresponds to a writer not processing S, i.e., WS = 0, and $[a_2 \wedge (b_1 \vee b_2)]$ corresponds to a writer processing S, i.e., WS = 1. The assertion q_1 listed in Table III reflects this case analysis: The global assertion c_1 describes the domain of the individual variables and is common to all assertions for the program. It was convenient to decompose the second disjunctive term into two disjunctive terms, b_1, b_2, corresponding to the reader processing Q and the reader *not* processing Q. A similar case analysis for the a_1 term is embedded in the conjunctive terms. Note that, as in problem 1, the prefixes W, R refer to writer and reader and the suffixes 1 and 2 refer to the upper and lower critical sections.

We will not burden the reader of the paper with a listing of the assertions at all points or with a proof of the various paths; the proof is quite similar to that illustrated for problem 1. However, it is of interest to abstract from q_1 sufficient information to prove the program's intent. For a discussion of deadlock the reader is referred to Reference 14.

As with problem 1 the decision concerning whether a requesting reader or a requesting writer gains access to the device is based on which one arrives first at the corresponding P(S) operation—not on arrival time of the readers and writers at the corresponding section entry points. This point is discussed in more detail below:

(1) The assertions indicate that any number of readers can share the device provided no writers are on, since if WD = 0, then from a_1 we see there are no constraints on RD. The assertions indicate that at most one writer is on the device because from observation of both a_1 and a_2 we note that WD = 0 or 1.

(2) The assertions indicate, as follows, that a reader's access to the device is not delayed provided no writers are processing S or are on the device, and provided no writers are pending on Q or S. The term a_1 indicates that if WS = WD = 0, i.e., no writers are processing S or on the device, and if WPENQ = WPENS = 0, i.e., no writers are pending on Q or S, then RPENS = RPENQ = 0,

TABLE III—Main Assertion for Courtois Problem 2

Global: (All variables ϵ $\mathrm{I)} \wedge (\mathrm{L} \leq 1) \wedge (\mathrm{M} \leq 1) \wedge (\mathrm{N} \leq 1) \wedge (\mathrm{Q} \leq 1) \wedge (\mathrm{S} \leq 1) \wedge (\mathrm{RC} \geq 0) \wedge (\mathrm{RD} \geq 0) \wedge (\mathrm{WC} \geq 0) \wedge (\mathrm{WD} \geq 0) \wedge (\mathrm{RPENS} \geq 0)$
$\wedge (\mathrm{WPENS} \geq 0) \wedge (\mathrm{RPENQ} \geq 0) \wedge (\mathrm{WPENQ} \geq 0) \wedge (\mathrm{RPENL} \geq 0) \wedge (\mathrm{RPENMI} \geq 0) \wedge (\mathrm{RPENM2]} \geq 0) \wedge (\mathrm{WPENN1} \geq 0) \wedge (\mathrm{WPENN2} \geq 0)$
$\wedge (\mathrm{RS} \geq 0) \wedge (\mathrm{WS} \geq 0)$
q_1: (Writer *not* processing S)
$(\mathrm{RS} = \mathrm{u}(-\mathrm{S}+1)) \wedge (\mathrm{WS} = 0) \wedge (\mathrm{WD} = 0) \wedge (\mathrm{WC} = \mathrm{u(WC)}) \wedge (\mathrm{WPENQ} = 0) \wedge (\mathrm{RPENQ} = 0) \wedge (\mathrm{RPENS} = 0) \wedge (\mathrm{u(WPENS)} = \mathrm{u(S)} - \mathrm{S})$
$\wedge (\mathrm{RD} = \mathrm{RC}) \wedge (\mathrm{u(Q)} = \mathrm{u}(1 - \mathrm{RC})) \wedge (\mathrm{WPENS} = \mathrm{WC}) \wedge (\mathrm{WPENN1} = \mathrm{u(N)} - \mathrm{N} \wedge (\mathrm{WPENN2} = 0) \wedge (\mathrm{RPENL} = \mathrm{u(L)} - \mathrm{L}) \wedge (\mathrm{u(L)}$
$= \mathrm{u(S))} \wedge (\mathrm{RPENM1} + \mathrm{RPENM2} = \mathrm{u(M)} - \mathrm{M}) \wedge (\mathrm{RPENMI} \leq \mathrm{RD})$
(Writer processing S)
$\{(\mathrm{WS} = 1) \wedge (\mathrm{RS} = 0) \wedge (\mathrm{WPENS} = 0) \wedge (\mathrm{RPENS} = -\mathrm{S}) \wedge (\mathrm{S} = -\mathrm{u}(-\mathrm{S})) \wedge (\mathrm{RPENQ} = 0) \wedge (\mathrm{WPENQ} = \mathrm{u(Q)} - \mathrm{Q}) \wedge (\mathrm{WPENN1}$
$+ \mathrm{WPENN2} = \mathrm{u(N)} - \mathrm{N}) \wedge (\mathrm{RC} = \mathrm{RD}) \wedge (\mathrm{RPENL} = \mathrm{u(L)} - \mathrm{L}) \wedge (\mathrm{RPENM1} = 0) \wedge (\mathrm{RPENM2} = \mathrm{u(M)} - \mathrm{M}) \wedge (\mathrm{L} \leq \mathrm{u(S+1))}\} \wedge \{[(\mathrm{Q} \leq 0)$
$\wedge (\mathrm{RC} \geq 1) \wedge (\mathrm{WD} = 0) \wedge (\mathrm{WC} = -\mathrm{Q}) \wedge (\mathrm{WPENN2} = 0)] \vee [(\mathrm{RC} = 0) \wedge (\mathrm{M} = 1) \wedge (\mathrm{WC} = \mathrm{WD} + \mathrm{WPENN1} + \mathrm{WPENQ}) \wedge (\mathrm{WD} = \mathrm{u(WD))}$
$\wedge (\mathrm{WD} \geq \mathrm{u(WPENQ))]}\}$

indicating that no reader is deferred by S or Q from access to the device.

The issue of writer priority will be handled by applying case analysis to q_1.

- RPENQ is always 0; thus a V(Q) operation can only grant access to a deferred writer, never to a reader.
- RS is 0 or 1; thus, at most, one reader is processing S. If RS = 1, then RPENS = 0 and WPENS = 0 or 1. This indicates that if a reader is processing S, the subsequent V(S) operation can only signal a deferred writer.
- If WS = 1 then WPENS = 0, and there are no constraints on WPENQ. This indicates that all deferred writers are pending on Q (or N as discussed below), and since RPENQ = 0 a writer must get access to the device either immediately if RD = WD = 0, or when the next V(Q) is performed by either a writer or a reader.

As we mentioned above, the issue of granting access to a writer or a reader is determined by the arrival time at P(S). If this is indeed the intent of the program, then the above discussion serves to prove the correctness of the program. However, there are several important exceptions that deserve discussion. For example, while a writer is pending on S, all subsequent requesting writers will be deferred by N. Now these writers should be granted access to the device before any requesting readers receive it, which will be the situation under "normal" timing conditions. The deferred writer, at point ⑬, will be scheduled by a reader doing V(S), in which case the writer will perform V(N) and in turn will schedule a deferred writer. These previously deferred writers will not get blocked by S but will pass to P(Q). Of the readers requesting access, one will be blocked by S and the remainder by L. The only way this scheduling would not occur as stated would be if the deferred writer at point ⑬ passed through the writer section and performed a V(S) operation, thus scheduling a deferred reader before a writer processing the upper critical section could get through the first two instructions.

Exact calculation of computer network reliability

by E. HÄNSLER

IBM Research Laboratory
Ruschlikon, Switzerland

G. K. McAULIFFE

IBM Corporation
Dublin, Ireland

and

R. S. WILKOV

IBM Corporation
Armonk, New York

INTRODUCTION

The exact calculation of the reliability of the communication paths between any pair of nodes in a distributed computer network has not been feasible for large networks. Consequently, many reliability criteria have been suggested based on approximate calculations of network reliability. For a thorough treatment of these criteria, the reader is referred to the book and survey paper by Frank and Frisch[1,2] and the recent survey paper by Wilkov.[3]

Making use of the analogy between distributed computer networks and linear graphs, it is noted that a network is said to be connected if there is at least one path between every pair of nodes. A (minimal) set of links in a network whose failure disconnects it is called a (prime) link cutset and a (minimal) set of nodes with the same property is called a (prime) node cutset. If a node has failed, it is assumed that all of the links incident to that node have also failed. A cutset with respect to a specific pair of nodes n_s and n_t in a connected network, sometimes called an *s-t* cut, is such that its removal destroys all paths between nodes n_s and n_t.

The exact calculation of $P_c[s, t]$, the probability of successful communication between any pair of operative computer centers n_s and n_t, requires the examination of all paths in the network between nodes n_s and n_t. More specifically, if each of the n nodes in any given network fail with the same probability q and each of the b links fail with the same probability p, then $P_c[s, t]$ is approximately given by

$$P_c[s, t] = \sum_{i=0}^{b} A^e_{s,t}(i)\,(1-p)^i p^{b-i}, \quad p \gg q. \tag{1}$$

In Eq. (1), $A^e_{s,t}(i)$ is the number of combinations of i links such that if only they are operative, there is at least one communication path between nodes n_s and n_t. On the other hand, the calculation of the probability $P_f[s, t]$ of a communication failure between nodes n_s and n_t requires the examination of all *s-t* cuts. For specified values of p or q, $P_f[s, t]$ is approximately given by

$$P_f[s, t] = \sum_{i=0}^{b} C^e_{s,t}(i)\,p^i (1-p)^{b-i}, \quad p \gg q. \tag{2}$$

For $q \gg p$, a similar expression can be given replacing $C^e_{s,t}(i)$ by $C^n_{s,t}(i)$. The coefficients $C^e_{s,t}(i)$ and $C^n_{s,t}(i)$ denote the total number of link and node *s-t* cuts of size i. The enumeration of all paths or cutsets between any pair of nodes n_s and n_t is not computationally possible for very large networks.

RELIABILITY APPROXIMATION BASED ON CUTSET ENUMERATION

If any network G of b links and n nodes, it is easily shown that the order of the number of cutsets is 2^{n-2}

whereas the order of the number of paths between any pair of nodes is 2^{b-n+2}. For networks having nodes of average degree (number of incident links) greater than four, $b>2n$ and $2^{b-n+2}>2^{n-2}$. Consequently, such networks have a larger number of paths than cutsets. Computation time would clearly be reduced in such cases by calculating network reliability from cutsets instead of paths. In this case $P_c[s, t]$ can be obtained from $P_c[s, t]=1-P_f[s, t]$, where $P_f[s, t]$ can be calculated from Eq. (2). Alternatively,

$$P_f[s, t]=P\left[\bigcup_{i=1}^{N} \underline{C}^{i}_{s,t}\right] \tag{3}$$

where $\underline{C}^{i}_{s,t}$ is the event that all links fail in the ith prime s-t cut and N is the total number of prime cutsets with respect to nodes n_s and n_t. The calculation of $P_f[s, t]$ from Eq. (2) clearly requires the examination of all s-t cuts. The number of prime s-t cuts is usually much smaller. However, $P_f[s, t]$ is not readily calculated from Eq. (3) because the $\underline{C}^{i}_{s,t}$ are not mutually exclusive events.

Following Wilkov,[4] we shall use $P_f=\text{Max}_{s,t}P_f[s, t]$ as an indication of the overall probability of service disruption for a given computer network. For specified values of p or q, P_f depends only on the topology of the network. A maximally reliable network clearly has a topology which minimizes P_f and hence minimizes $\text{Max}_{s,t}C^{n}_{s,t}(m)$ or $\text{Max}_{s,t}C^{e}_{s,t}(m)$ for small (large) values of m when p or q is small (large). Letting $X^{n}_{s,t}(m)$ and $X^{e}_{s,t}(m)$ denote the number of prime node and edge s-t cuts of size m, $X^n(m)=\text{Max}_{s,t}X^{n}_{s,t}(m)$ and $X^e(m)=\text{Max}_{s,t}X^{e}_{s,t}(m)$ have been proposed[4] as computer network reliability measures. These measures $X^n(m)$ and $X^e(m)$ denote the maximum number of prime node and edge cutsets of size m with respect to any pair of nodes. A maximally reliable network is such that $X^n(m)$ and $X^e(m)$ are as small as possible for small (large) values of m when the probability of node or link failure is small (large).

In the calculation of $X^n(m)$ and $X^e(m)$ for any given network, all node pairs need not be considered if all nodes or links have the same probability of failure. It has been shown[5] that in order to calculate $X^n(m)$ and $X^e(m)$, one need only consider those node pairs whose distance (number of links on a shortest route between them) is as large as possible. For a specified pair of nodes n_s, n_t, $X^{e}_{s,t}(m)$ can be calculated for all values of m using a procedure given by Jensen and Bellmore.[6] Their procedure enumerates all prime link cutsets between any specified pair of nodes in a non-oriented network (one consisting only of full or half duplex links). It requires the storage of a large binary tree with one terminal node for each prime cutset. Although these cutsets are not mutually exclusive events, it has been suggested[6] that Eq. (3) be approximated by

$$P_f[s, t]\approx\sum_{i=0}^{N} P[\underline{C}^{i}_{s,t}]. \tag{4}$$

However, it is shown in the following section that no additonal computation time is required to actually compute $P_f[s, t]$ exactly.

EXACT CALCULATION OF COMPUTER NETWORK RELIABILITY

A simple procedure is described below to iteratively calculate a minimal set of mutually exclusive events containing all prime link s-t cuts. This procedure starts with the prime cutset consisting of the link incident to node n_t. Subsequently, these links are re-connected in all combinations and we then cut the minimal set of links adjacent to these that lie on a path between node n_s and n_t, assuming that the network contains no pendant nodes (nodes with only one incident link). The link replacements are iterated until the set of links connected to node n_s are reached. The procedure is easily extended to provide for node cutsets as well and requires a very small amount of storage since each event is generated from the previous one. $P_f[s, t]$ is obtained by accumulating the probabilities of each of the mutually exclusive events.

Procedure I

1. Initialization

Let: N be the set of all nodes except nodes n_s.
C be the set of all links not incident to node n_s.
$M_1=\{n_s\}$
F_1 be the set of links incident to both n_s and n_t
S_1 be the set of links incident to n_s but not n_t
b_1 be a binary number consisting of only $|S_1|$ ones
$i=1$

2. Let:

T_i be a subset of S_i consisting of those elements in S_i for which the corresponding digit in b_i is unity.
M_{i+1} be a subset of N consisting of nodes incident to the links in T_i.
$N = N-M_{i+1}$.
F_{i+1} be a subset of C consisting of links incident to n_t and adjacent to any member of T_i.
S_{i+1} be a subset of C consisting of links incident to nodes in N other than n_t and adjacent to any member of T_i.
$C = C-(S_{i+1}\cup F_{i+1})$.

```
3. If $S_{i+1} \neq \emptyset$, then let:
   $b_{i+1}$  be a binary number with $| S_{i+1} |$ ones
   $i = i+1$
             Go to step 2
   Otherwise, let:
   $T_{i+1} = \emptyset$
   $CS = \bigcup_{k=1}^{i+1} [F_k \cup \bar{T}_k \cup (S_k - T_k)]$,
         where $CS$ is a modified cutset and $\bar{T}_k$ indicates
         that the links in set $T_k$ are connected.
4. Let:
   $C = C \cup F_{i+1} \cup S_{i+1}$
   $N = N \cup M_{i+1}$
   $b_i = b_i - 1$ (modulo 2)
   If $b_i < 0$, go to step 5. Otherwise, go back to step 2.
5. Let $i = i-1$. If $i \neq 0$, go back to step 4. Otherwise,
   terminate the procedure.
```

In the calculation of $P_f[s, t]$, Procedure I performs a depth first search of the given network starting at node n_s and traversing several links at the same time. The index i indicates how far from n_s the search has progressed and b_i indicates the links traversed at the ith level of the search. During the search, set N keeps track of the nodes which have not yet been reached and C is the set of links not yet traversed. At the ith level, set F_{i+1} is a subset of the links not yet traversed which are incident to node n_t and hence must be disconnected in the formation of an s-t cut. Set S_{i+1} consists of edges in C which lie on a path to n_t but which need not necessarily be disconnected in the formation of an s-t cut. The edges in $T_{i+1} \subseteq S_{i+1}$ are those which are connected as we traverse the network toward node n_t. When set S_{i+1} is empty, the edges incident to n_t have been reached and this portion of the search is terminated with the formation of a modified s-t cut in step 3 of the procedure. The modified s-t cut is actually a group of states in the network or an event in which all links in an s-t cut are disconnected and the links in all T_i in this part of the search are connected. It is the set of connected links which makes this modified s-t cut mutually exclusive of all of the modified s-t cuts previously generated during the execution of Procedure I. In step 4, we back track one level and then continue the search by traversing a different subset of the links in S_i. After all combinations of the links in S_i have been traversed, we back track one additional level and the search continues with traversal of a different combination of the links in S_{i-1}. The procedure terminates when we have back tracked all the way up to node n_s. It is shown in the proof of the following theorem that the modified s-t cuts generated are mutually exclusive and collectively exhaustive.

Theorem I:

Procedure I generates a collectively exhaustive set of mutually exclusive modified s-t cuts.

Proof:

Part I—Prime s-t cuts

In this part of the proof, it is shown that every modified cutset CS generated in step 3 of Procedure I contains a prime s-t cut. We begin by noting that the links in $T_k (1 \leq k \leq i+1)$, traversed in the depth first search through the given network, form subnetworks containing node n_s. For any such subnetwork, set M_i contains the nodes at a distance of i from n_s. Each modified cutset CS generated by the procedure consists of all links in such a subnetwork being connected and all links in $S_k - T_k$ and F_k, that connect nodes inside the subnetwork with those outside the subnetwork, being disconnected. Node n_t is never contained in the subnetwork since any link incident to n_t must be contained in some set F_k and is therefore always disconnected.

Part II—Mutually exclusive

In order to show that the modified cutsets obtained from Procedure I are mutually exclusive, we shall demonstrate that every pair of modified cutsets disagree in the state of at least one link appended at some level j. Specifically, for any pair of distinct modified cutsets CS_p and CS_q, there exists a value of j for which if $\bar{T}_k \subset CS_p$ and $\bar{T}_k' \subset CS_q$, then $T_k = T_k'$ for $k \leq j-1$ but $T_j \neq T_j'$. This implies that during the generation of CS_p and CS_q from Procedure I, $b_k (k \leq j-1)$ was the same in both cases but the value of b_j was different. Otherwise, if CS_p and CS_q were generated from the same values of b_k for all k, then CS_p and CS_q would be identical. It is now noted that if $T_j \neq T_j'$, there exists a link e such that $e \in T_j$ and $e \notin T_j'$ which implies that $e \in (S_j - T_j')$. It follows that link e appears connected in CS_p and disconnected in CS_q.

Part III—Collectively exhaustive

We shall prove that the modified s-t cuts obtained from Procedure I are collectively exhaustive by showing that every state of the network in which nodes n_s and n_t cannot communicate is contained in one of the modified s-t cuts. We proceed by noting that in any given state of the network that includes an s-t cut, there is a maximal set of nodes N_s connected to node n_s which does not include node n_t. If set N were discarded in Procedure I, then the resulting modified s-t cuts would contain every state of the links on set N_s in which there is a path between every pair of nodes in

N_s. This follows from the fact that in the generation of all modifications of the same prime *s-t* cut, the deletion of set N from Procedure I would result in set S_k for all k containing every link on set N_s. As we sequence through all b_k, these links would be connected and disconnected in Procedure I in every combination in the traversal of all paths from n_s to every other node in N_s. It is now noted that any modified *s-t* cut generated from Procedure I that includes a connected subnetwork on N_s specifies as cut all links connecting nodes N_s to nodes in $N-N_s$, where N is the set of all nodes in the network. All links in the network connecting pairs of nodes in $N-N_s$ would be unspecified.

Taking advantage of the unspecified links, it is possible to extend one of the modified *s-t* cuts generated by Procedure I with set N deleted to match any specified state of the network in which nodes n_s and n_t are not connected. The effect of using set N in Procedure I is to omit several links from many of the S_k. Significantly fewer modified *s-t* cuts are thereby generated since the states of the redundant links joining pairs of nodes in N_s would not be specified. However, these modified *s-t* cuts clearly include all of those generated when set N is neglected. This is evident since each of the links on N_s not specified can be assigned a particular state in order to match a given modified *s-t* cut obtained from Procedure I with set N omitted. Consequently, any specified state of the network containing an *s-t* cut is included in one of the modified *s-t* cuts obtained from Procedure I. Q.E.D.

It should be noted that the collectively exhaustive set of mutually exclusive modified *s-t* cuts obtained from Procedure I is not minimum. This is due to the fact that for any prime *s-t* cut, Procedure I as given generates too many subnetworks on the set of nodes N_s connected to n_s. However, Procedure I is easily modified to eliminate the generation of any subnetworks on N_s that contain circuits. This is done by eliminating all T_i in step 2 of the procedure in which two or more links are incident to the same node. The formation of any other circuits in subnetworks on N_s is avoided through the use of set N in Procedure I. The result is that the connected links in any modified *s-t* cut would form trees on N_s.

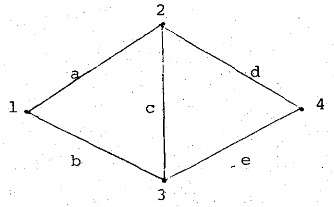
It is noted that in the procedure given above, nodes have been assumed to be perfectly reliable. However, Procedure I can also be applied in the case that nodes fail and links are perfectly reliable. In the event that nodes and links may fail simultaneously, assuming that their failures are statistically independent, following Hänsler[7] we can easily modify Procedure I to obtain a collectively exhaustive set of mutually exclusive modified *s-t* cuts consisting of nodes and links. We would proceed by introducing a binary number b^n_i consisting of only M_i ones for each of the sets M_i in Procedure I. Analogous to T_i, in step 2 we form a set T^n_{i+1} consisting of the nodes in M_{i+1} for which the corresponding digit in b^n_{i+1} is unity. F_{i+1} and S_{i+1} in step 2 would consist of links in C incident to nodes in T^n_{i+1}. Then any modified *s-t* cut CS formed in step 3 of Procedure I would consist of

$$CS=\bigcup_{k=1}^{i+1}[F_k\cup\bar{T}_k\cup(S_k-T_k)\cup T^n_k\cup(M_k-T^n_k)]\cup\{\bar{n}_t\} \qquad (5)$$

The only other *s-t* cut consists of node n_t being inoperative. The above modifications to Procedure I double the number of levels and therefore significantly increase the necessary computation time for any given network. However, the storage requirement of the modified procedure is still very small. A network of b links and n nodes would only require approximately $3b+2n$ words of storage to compute $P_f[s, t]$ in the presence of node and link failures. All modified cutsets are either printed out or their probabilities accumulated. Consequently, the exact calculation of $P_f[s, t]$ for any given network is limited only by the computer time required in view of the inherent computational complexity of the problem.

EXAMPLES OF NETWORK RELIABILITY CALCULATIONS

In this section, Procedure I will be used to obtain $P_f[s, t]$ for several networks, assuming that all nodes are perfectly reliable and all links fail with the same probability p. We shall first consider the simple network shown in Figure 1 in order to demonstrate the modified 1-4 cuts obtained from Procedure I. Figure 1



Prime 1-4 Cuts	Modified 1-4 Cuts
d e	$\bar{a}$ $\bar{b}$ d e
b c d	$\bar{a}$ b d e $\bar{c}$
a c e	$\bar{a}$ b d c
a b	a $\bar{b}$ d e $\bar{c}$
	a b

Figure 1—Example illustrating the calculation of node pair failure probability

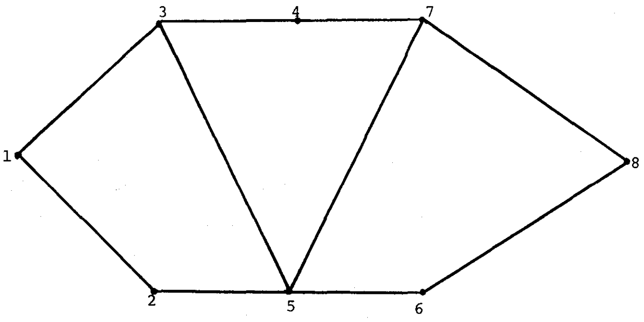


Figure 2—Example for comparison of approximate and exact reliability calculations

shows the four prime cutsets between nodes 1 and 4, which are not mutually exclusive. Also listed there are the six mutually exclusive modified 1-4 cuts obtained from Procedure I in the order in which they are obtained. Note that the second and fourth modified 1-4 cuts are not prime since link *a* or *b* has been disconnected in order for the corresponding events to be mutually exclusive.

The network shown in Figure 2 has been given by Jensen and Bellmore[6] as an example of their procedure for enumerating all prime cutsets with respect to a given pair of nodes. They listed 16 prime 1-8 cuts for the network of Figure 2. From these cutsets, $P_f[1,8]$ was approximated by

$$P_f[1,8] \approx 4p^2+8p^3+4p^4 \qquad (6)$$

However, from the mutually exclusive modified 1-8 cuts obtained from Procedure I, $P_f[1,8]$ is actually given by

$$P_f[1,8]=4p^2+6p^3-16p^4-32p^5+115p^6-134p^7 +79p^8-24p^9+3p^{10} \qquad (7)$$

It is clear from this example that the approximation to $P_f[s,t]$ given by Jensen and Bellmore[6] is reasonable

TABLE I—Polynomial Coefficients for P_f [9, 6] for Networks of Figure 3

Coefficient	Figure 3a	Figure 3b
c_2	3	2
c_3	6	4
c_4	−6	9
c_5	−25	−22
c_6	−25	−153
c_7	237	572
c_8	−417	−874
c_9	364	744
c_{10}	−177	−371
c_{11}	46	102
c_{12}	−5	−12

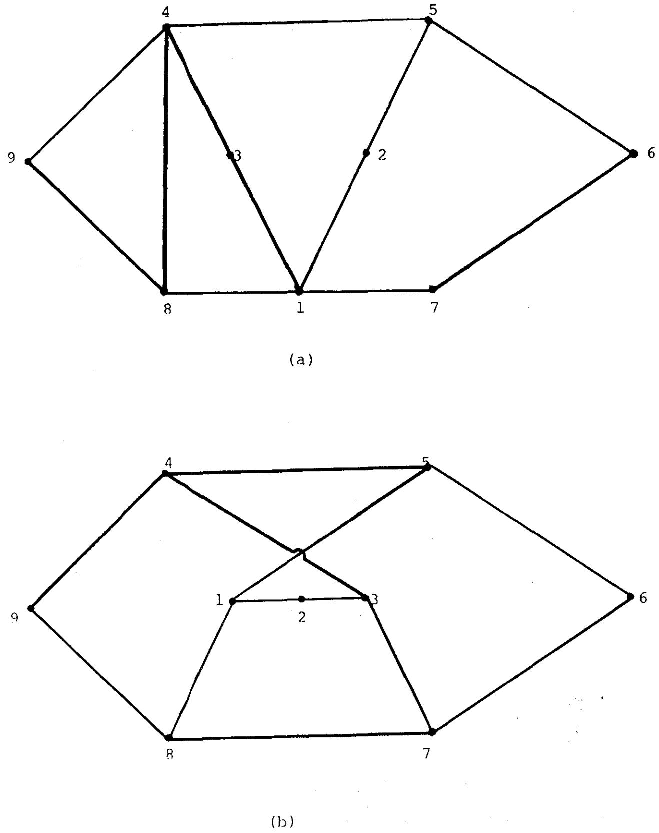


Figure 3—ARPA subnetwork topologies having 9 nodes and 12 links: (a) actual, (b) example based on $X^e(m)$

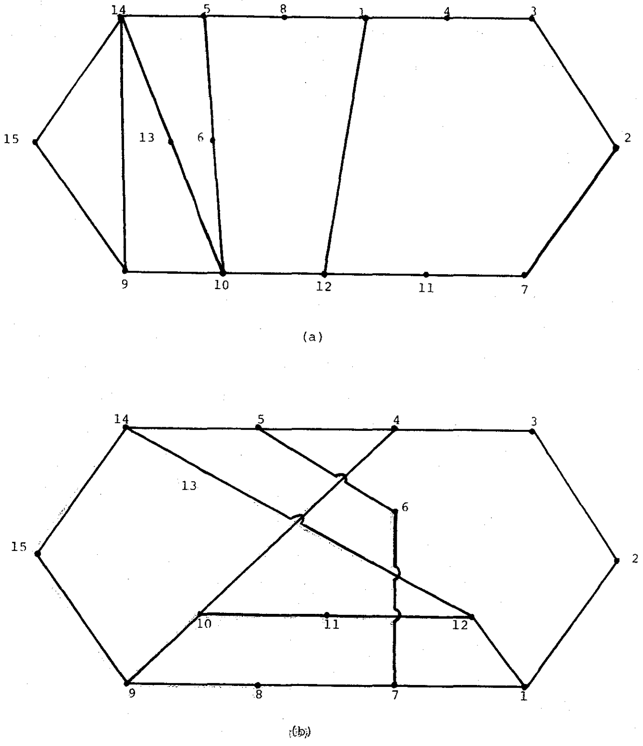


Figure 4—ARPA subnetwork topologies having 15 nodes and 19 links: (a) actual, (b) example based on $X^e(m)$

TABLE II—Polynomial Coefficients for P_f [15, 2] for Networks of Figure 4

Coefficient	Figure 3a	Figure 3b
c_2	12	3
c_3	5	6
c_4	−56	28
c_5	55	7
c_6	−84	−620
c_7	701	−1267
c_8	−521	20379
c_9	−6212	−77855
c_{10}	24039	171797
c_{11}	−46457	−257512
c_{12}	58369	279128
c_{13}	−51647	−224691
c_{14}	33123	135228
c_{15}	−15433	−60303
c_{16}	5121	19402
c_{17}	−1152	−4269
c_{18}	158	576
c_{19}	−10	−36

for only very small values of p since only the first coefficient in that approximation is exact.

Two topologies for 9 node and 15 link subnetworks of the ARPA network are shown in Figure 3. The network shown in Figure 3a was given by Frank, et al.[8] Figure 3b is a maximally reliable network based on $X^n(m)$ and $X^e(m)$ for small m obtained by Wilkov.[9] Assuming all nodes are perfectly reliable and all links fail with the same probability p, $P_f[9, 6]$ can be expressed as

$$P_f[9, 6] = \sum_{i=2}^{12} c_i p^i. \tag{8}$$

The coefficients in Eq. 8 for Figures 3a and 3b are listed in Table I. They have been obtained in 18 seconds using an APL implementation of Procedure I on a 360 model 91 computer. Consistent with the results in Reference 9, Figure 3b has smaller coefficients than Figure 3a for small powers of p. Furthermore, we have found that there are a total of 2,772 cutting states with respect to nodes 9 and 6 in Figure 3b compared with 3,011 in Figure 3a. Similar results have been obtained for the 15 node and 19 link ARPA subnetwork topologies shown in Figure 4. The topology shown in Figure 4a was given by Frank, et al.[8] and Figure 4b was obtained by Wilkov[9] based on $X^n(m)$ and $X^e(m)$. The polynomial coefficients for $P_f[15, 2]$ are given in Table II. The total number of cutting states between nodes 15 and 2 is 49.7 thousand for Figure 4a and 44.9 thousand for Figure 4b.

CONCLUSION

A procedure has been given for calculating the node pair failure probability in computer networks exactly, using little more computation time than previously required to obtain an upper bound on $P_f[s, t]$. Furthermore, the storage requirement of the given procedure grows only linearly with the number of links in the given network. Unfortunately, due to the inherent computational complexity of the problem, the necessary computation time grows exponentially with the size of the given network. Nonetheless, it has been found to be computationally feasible to use the procedure given herein for networks as large as the ARPA network.

REFERENCES

1 H FRANK I T FRISCH
Communication, transmission, and transportation networks
Addison-Wesley Publishing Company Reading
Massachusetts 1971

2 H FRANK I T FRISCH
Analysis and design of survivable networks
IEEE Transactions on Communication Technology
Vol COM-18 1970 pp 501-519

3 R S WILKOV
Analysis and design of reliable computer networks
IEEE Transaction on Communications Vol COM-20
June 1972 pp 660-628

4 R S WILKOV
Reliability considerations in computer network design
Proceedings of IFIP Congress '71 Ljubljana Yugoslavia
August 1971

5 R S WILKOV
On the design of maximally reliable communication networks
Proceedings of the Sixth Annual Princeton Conference on
Information Sciences and Systems March 1972

6 P A JENSEN M BELLMORE
An algorithm to determine the reliability of a complex system
IEEE Transactions on Reliability Vol R-18 1969 pp 169-174

7 E HÄNSLER
A fast recursive algorithm to calculate the reliability of a communication network
IEEE Transactions on Communications Vol COM-20
June 1972 pp 637-640.

8 H FRANK et al
Store and forward computer networks
Third Semiannual Technical Report for ARPA Contract
DAHC 15-70-C-0120 June 1971

9 R S WILKOV
Design of computer networks based on a new reliability measure
Proceedings of the International Symposium on
Computer-Communication Networks and Teletraffic
Polytechnic Institute of Brooklyn New York April 1972

A framework for hardware-software tradeoffs in the design of fault-tolerant computers

by K. M. CHANDY, C. V. RAMAMOORTHY and A. COWAN

The University of Texas at Austin
Austin, Texas

INTRODUCTION

The theory of fault-tolerant computer design has developed rapidly. Several techniques using hardware or software have been suggested. A student is often faced with the problem of developing a common perspective for a variety of methods. In this paper we attempt to develop a simple framework within which different methods can be compared. We use a set of very elementary indices to construct the framework. The indices are quite crude and our framework is somewhat ad hoc. Though a unified theory would be extremely useful we have not attempted to develop one here. Our discussion is a first pass at identifying some goals of reliable design and an attempt at quantifying some parameters. We discuss only a very small set of the techniques that have been proposed for fault-tolerant computers. Methods for constructing relevant indices for these techniques are presented. We feel that these indices are relevant for most reliability techniques.

We shall classify all techniques for achieving reliability into two categories: hardware techniques and software techniques.

In the following discussion of reliability we consider an aerospace system such as a missile interception system or an air-traffic control system. The system has a specific mission which should be accomplished in a specified amount of time. A (large) penalty is incurred if the system does not accomplish its mission. We shall refer to this penalty as the cost of mission failure. A *lateness penalty* is incurred if the time taken to accomplish the mission exceeds the specified time. The longer the time taken to complete the mission, the greater the lateness penalty. Different methods for improving reliability are evaluated with such a system in mind.

Our approach to reliability rests on a framework of four indices called the Hardware Reliability Efficiency index (HRE), the Software Reliability Efficiency index (SRE), the Real-Time Criticality index (RTC) of a system, and the inclusion factor. For a given method of achieving reliability HRE and SRE are measures of the increase in reliability of the system per unit of expenditure. For the same amount of expenditure, a method with a high HRE (or SRE) gives better reliability than a method with low HRE (or SRE). In this paper we shall discuss ways of computing the efficiency indices for several different reliability methods. The real-time criticality index is a measure of the penalty incurred for a late completion of the system mission. Thus an air-traffic control system would have a high RTC compared to other systems. The inclusion factor (defined later) is a dimensionless number; if the inclusion factor for a given method is less than one, then that method should not be used in the system. The inclusion factor is a function of the method being considered and of system objectives. Thus a given technique may be optimally included in the design of one system and excluded from another.

We shall now discuss each of the indices in turn.

Hardware reliability efficiency index

Several models[1-5] have been constructed for designing reliable machines from intrinsically less reliable components by using redundant components: we shall refer to these methods as hardware methods. The "cost" of a hardware method is the dollar amount required to buy or build the redundant hardware. We may define the Hardware Reliability Efficiency index, (HRE), of a hardware method as the incremental increase in reliability (defined in some appropriate manner) per incremental increase in the amount spent on purchasing redundant hardware.

Software reliability efficiency index

Methods have recently been devised to improve reliability primarily by means of software.[5-8] In one such method, when an error is detected, the system is "rolled back" to an error-free state, which was saved earlier, and computation is restarted from that point. Fault-tolerance is achieved in this case at the expense of the time required to rollback and to reprocess to the point of error. This is discussed later, in greater detail. Deadlock prevention methods[5,6] are also examples of improving the reliability of a system at the expense of a reduced rate of utilization of system resources. We shall call these methods of achieving reliability *software* methods. The "cost" associated with software methods is generally the additional *time* required for processing (with these methods). The capital cost associated with developing the software may sometimes be neglected. An index of Software Reliability Efficiency (SRE) is the incremental increase in reliability per unit of additional time spent in achieving this improved reliability. In summary, reliability is achieved in hardware methods by spending more money, while in software methods reliability is achieved at the expense of processing time. When the capital cost of software methods cannot be ignored a combination of HRE and SRE is used.

Real-time criticality index

In some systems, software methods have to be ruled out, since the time available to complete the mission is too short to permit methods which require additional time. In other cases, the longer the system takes to complete a mission, the more expensive the consequences. This is typically the case in a missile interception system. A useful index is the Real-Time Criticality index (RTC) which is the cost incurred per unit delay in completing the system's mission. RTC will be high in many aerospace applications and comparatively low in some commercial systems. RTC is the penalty rate for late mission completion.

Inclusion factors

The indices HRE, SRE, and RTC, together with the penalty incurred if the mission fails indicate the methods to be selected for the design from the set of methods available. We shall define the *inclusion factor* for a hardware method as:

$$\text{HRE} \times \text{penalty of mission failure}$$

and the inclusion factor for a software method as:

$$\frac{\text{SRE}}{\text{RTC}} \times \text{penalty of mission failure.}$$

The inclusion factors are dimensionless. The inclusion factor is the ratio of the decrease in expected cost of mission outcome and the cost incurred in achieving this decrease. If the decrease in expected cost of mission outcome is less than the cost incurred in using a method, then that method should not be used in the system. In other words, when designing a system, a designer may exclude from consideration all methods with inclusion factors less than one. In some aerospace applications though the penalty of failure is high, the real-time criticality index is so large that the inclusion factor for software methods is less than one, and hence these methods need not be considered.

It is possible that the indices HRE and SRE may be interdependent: the index for a hardware method may depend on whether a software method has been implemented. Furthermore, the additional costs associated with implementing a method may not be continuous, but may increase in discrete amounts.

We shall now study a few methods for improving reliability and discuss techniques for computing hardware and software reliability indices for these methods. The real-time criticality index and the cost of failure depend on the system rather than on the design used and hence will not be discussed further.

SOFTWARE METHODS

Rollback

Discussion

In many real-time systems it is necessary to recover rapidly from a transient error. One way of achieving quick recovery is to "rollback" the program when a transient error is detected[5,8] and to restart the program at a previously saved error-free state. The state of a program refers to the content of relevant areas in memory, to the content of registers, and to all other relevant information necessary to restart the program at that point. If a transient error is detected, and if an error-free state of the program has not been saved earlier, the program will have to be restarted at the very beginning, resulting in slow recovery. On the other hand, if recovery is to be quick, error-free states of the program will have to be saved very frequently resulting in large overhead. Thus there is a tradeoff between recovery time and overhead: the quicker the recovery time, the larger the overhead.

Chandy and Ramamoorthy[8] have discussed the problem of determining the optimum points in a program at which the state of the program ought to be saved. They suggested a technique for minimizing the overhead given the maximum allowable recovery time. The overhead is the time spent in saving states of the program.

We shall briefly review the rollback design suggested in Reference 8 and then compute the software reliability efficiency index for this design.

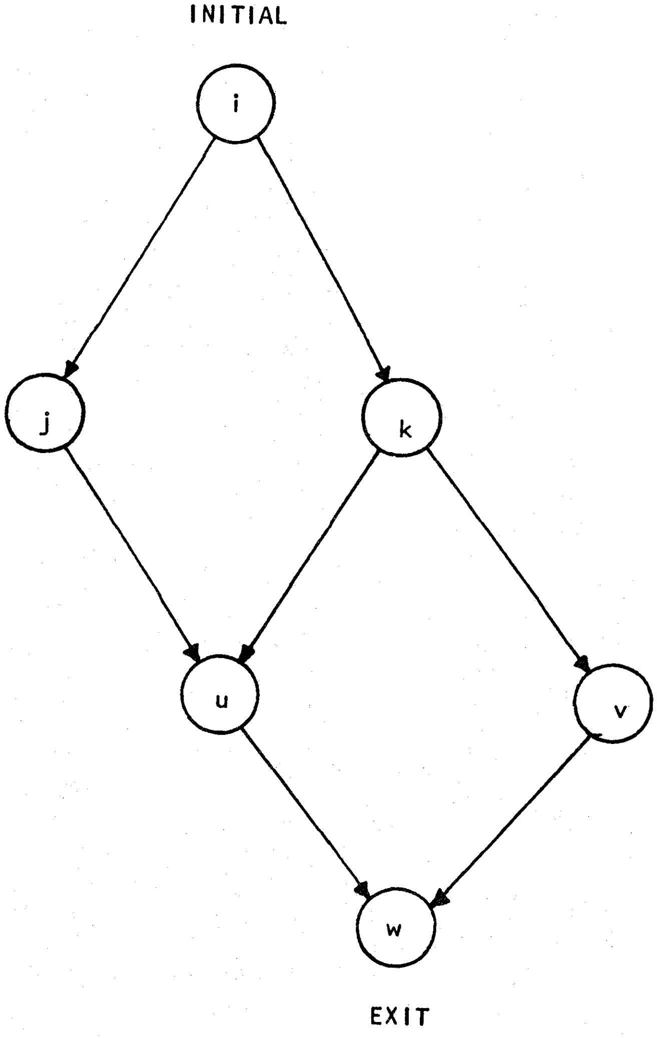


Figure 1

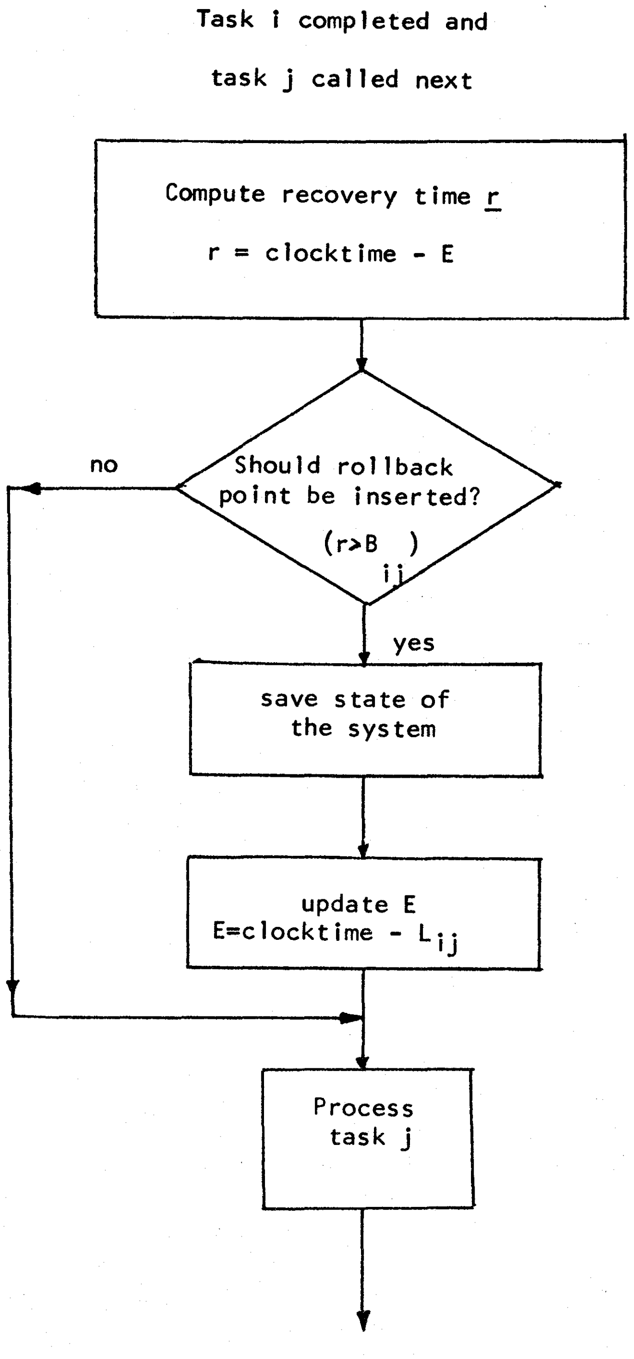


Figure 2

The objective of the design is to determine the optimum points at which states of the system should be saved so as to minimize overhead (i.e., time spent in saving states) subject to the constraint that the recovery time should not exceed some given value M.

The locus of control of a program may be represented by a directed graph where a vertex in the graph corresponds to a task in the program; an edge from vertex i to vertex j exists if and only if control may pass (with non-zero probability) to task j after task i is completed. If there are edges from vertex i to vertices j and k, then control may pass from task i to either task j or task k; see Figure 1. A task consists of an arbitrary set of instructions. If a transient error is detected during the processing of a task, the program is rolled back to a previously saved state (or if none exists to the very beginning). If no error is detected, a short "detection routine" may be run to check key variables and again rollback is employed if an error is detected. On the other hand, if no error is detected, the state of the program is assumed to be error free. The state of a program may be saved only after a task is finished and before another task is begun.

Let L_{ij} be an estimate of the time taken to load a state of the system which was saved after task i was finished and if task j was called next. At any point P in the program, let r be the recovery time (i.e., the time taken to load the most recently saved state and to recompute from this saved state to point P). It is shown in Reference 8 that there exist numbers B_{ij} such that the optimal decision is to save the state of the system after task i is completed and if task j is to be processed next, if and only if $r>B_{ij}$. A flow-chart for determining whether a state ought to be saved after task i is finished and if task j is called next is shown in Figure 2.

Computing the software reliability efficiency index

Let T be the time required to complete the program if there is no error, and without implementing a rollback method. Let H be the overhead incurred by implementing a rollback procedure. H can be easily computed for an arbitrary program as shown in Reference 8. Recollect that the rollback procedure is designed so that the maximum recovery time will not exceed a given value M. If the mission is completed in $T+S$ units rather than T units a "lateness penalty" is incurred which gets larger as S increases. We shall find the reliability of a system with rollback as a function of S, the amount of "lateness" permitted. We shall assume that failures occur according to the exponential failure law, and the mean time between failures is $1/a$.

If $S=0$ then the program must finish in T time units without error. The probability of no error in T time units is e^{-aT}. Letting $R(S)$ be the reliability, defined as the probability of completing a successful mission, we have:

$$R(0)=e^{-aT}$$

If $S=H+M$, then it is possible to implement rollback and to allow recovery from one error by means of rollback. The reliability in this case is the probability of no error in $T+H$ time units (in which case no rollback is necessary) plus the probability of exactly one error in $T+H$ units followed by a period of M error free units in which recovery is taking place.

$$R(H+M)=e^{-a(T+H)}+\frac{[a(T+H)]^1e^{-a(T+H+M)}}{1!}$$

By the same argument, if $S=H+2M$ then two error recoveries are possible and

$$R(H+2M)=R(H+M)+\frac{[a(T+H+M)]^2e^{-a(T+H+2M)}}{2!}$$

In general

$$R(H+nM)=R(H+(n-1)M)+\frac{\{a[T+H+(n-1)M]\}^n e^{-a(T+H+nM)}}{n!}$$

$$\text{for } n=2, 3, \ldots$$

If we are considering delaying the time required to complete the mission by S units we get the Software Reliability Efficiency index to be:

$$\text{SRE}=\frac{R(S)-R(0)}{S}$$

Note that in this analysis undetected and permanent errors were ignored. They can be included quite simply. Let $Q(S)$ be the probability of the event that there is no undetected or permanent error in S units and let it be independent of other events. Then we have

$$\text{SRE}=\frac{Q(S)\cdot[R(S)-R(0)]}{S}$$

Instructional retrial

If an error is detected while the processor is executing an instruction, the instruction could be retried, if its operands have not already been modified. This technique is an elementary form of rollback: recovery time never exceeds the execution time of an instruction, and overhead is negligible. However, there is a probability that an error will persist even after instruction retry. Let this probability be Q. The SRE for this technique can be computed in a manner identical to that for rollback and has the same form. The SRE for instruction retrial will in general be higher than that for rollback.

Deadlock prevention

Discussion

Prevention of deadlocks is an important aspect of overall system reliability. Deadlocks may arise when procedures refuse to give up the resources assigned to them, and when some procedures demand more resources from the system than the system has left unassigned. Consider a system with one unit each of two resources A and B, and two procedures I and II. Now suppose procedure I is currently assigned the unit of resource A while II is assigned B. Then if procedure I demands B and II demands A, the system will be in a deadlock: neither procedure can continue without the resources already assigned to the other. The hardware approach to this problem is to buy sufficient resources so that requests can be satisfied on demand.

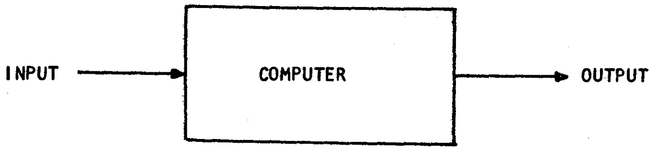
Habermann and others[6,7] have discussed methods for preventing deadlocks without purchasing additional resources. In these methods sufficient resources are always kept in reserve to prevent the occurrence of deadlock. This may entail users being (temporarily) refused resources requested, even though there are unassigned resources available. Keeping resources in reserve also implies that resource utilization is (at least temporarily) decreased. An alternative approach is to allocate resources in such a manner, that even though it is possible that deadlocks *might* arise, it is very improbable that such a situation could occur. The tradeoff here is between the probability of deadlock on the one hand and resource utilization (or throughput) on the other. The tradeoff is expressed in terms of the software reliability efficiency index.

Determining the software reliability efficiency index

The probability P of a deadlock while the mission is in progress and the time T required to complete the mission (assuming no deadlock) using a scheme where resources are granted on request are determined through simulation. The time $(T+H)$ required to complete the mission using a deadlock prevention scheme is also determined by means of simulation. If $Q(L)$ is the probability that no malfunctions other than deadlock arise in L time units, then assuming independence, we have:

$$\mathrm{SRE} = \frac{Q(T+H) - Q(T)\cdot(1-P)}{H}$$

At this time we know of no way of computing H and P analytically.



Simplex Configuration

Figure 3a

Summary of software methods

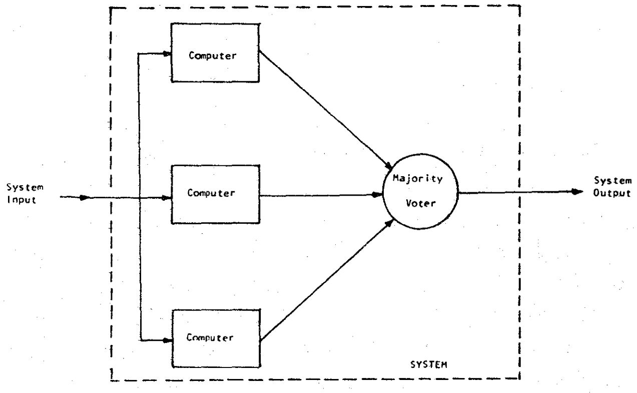
Different methods for improving the overall reliability of a system using software have been discussed. The software reliability efficiency index was suggested as an aid in evaluating software methods. Techniques for computing SRE were discussed. Similar techniques can be used for computing SRE for other software methods.

HARDWARE METHODS

Triple modulo redundancy

Discussion

Triple Modulo Redundancy (TMR) was one of the earliest methods[1] suggested for obtaining a reliable system from less reliable components. The system output (Figure 3) is the majority of three identical components. If only one of the components is in error, the system output will not be in error, since the majority of



TMR

Figure 3b

components will not be in error. Thus, the system can tolerate errors in any one component; note that these errors may be transient or permanent. In this discussion we discuss only permanent errors.

Computing the hardware reliability efficiency index

Let P be the probability that a permanent malfunction occurs in a given component before the mission is completed. If failures obey an exponential law, and if the average time to a failure is $1/a$, then $P=1-e^{-aT}$, where T is the time required to complete the mission. If the system is a discrete transition system (such as a computer system), then the time required to complete the mission can be expressed as N cycles (iterations) where computation proceeds in discrete steps called cycles. If the probability of failure on any cycle is p independent of other cycles then

$$P=1-(1-p)^N$$

Let v be the probability of a malfunction in the vote-taker before the mission is complete independent of other events. The reliability R of a TMR system is the probability that at least two components and the vote-taker do not fail for the duration of the mission.

$$R=[(1-P)^3+3(1-P)^2\cdot P]\cdot(1-v)$$

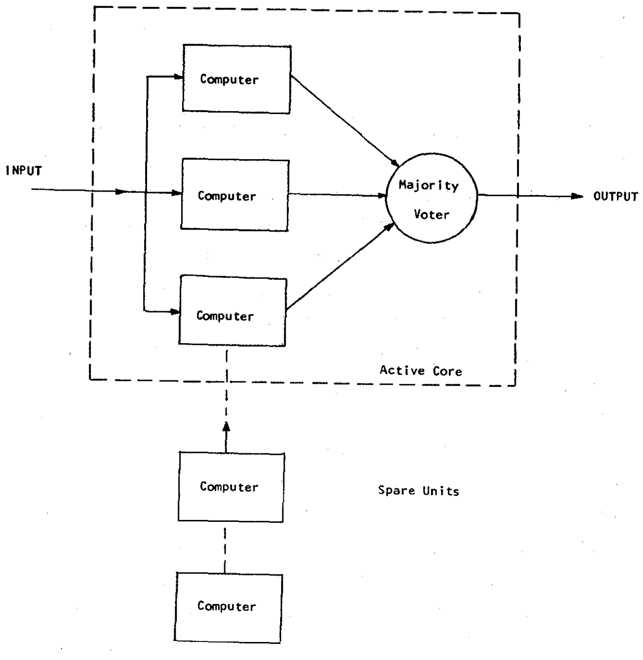
If C is the cost of each component, and D the cost of the vote-taker, the hardware reliability efficiency index is:

$$\text{HRE}=\frac{[(1-P)^3+3(1-P)^2\cdot P]\cdot(1-v)-(1-P)}{2C+D}$$

Transient errors can also be included quite easily in HRE.

Hybrid system

Discussion



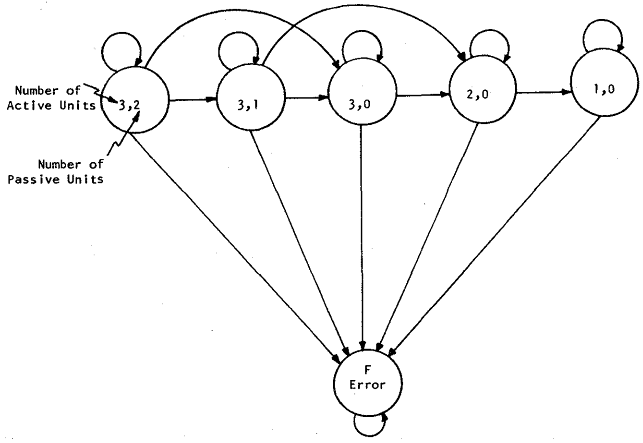
Hybrid System (5,3)

Figure 4

Mathur and Avizienis[2] discuss an ingenious method of obtaining reliability by using TMR and spares, see Figure 4. The spares are not powered-on and will be referred to as "inactive" components. If at any point in time, one of the three active components disagrees with the majority, the component in the minority is switched out and replaced by a spare. The spare must be powered-up and loaded; one method of loading the component is to use rollback and load the component with the last saved error-free state, and begin computation from that point. If at most one component fails during a cycle and if the vote-taker is error-free, this system is fail-safe until all the spares are used up, i.e., the system output will not be in error. Consider a comparison of a system with three active units and two spares with another system which has five active units. If at most one unit can fail at a time then the majority is always right and the system with three active units is at least as good as a system with five active units (since a majority of two active units is as right as a majority of four). Thus if at most one unit fails at a time, the number of active units need never exceed three; additional units should be kept as spares. Of course in digital computer systems where computation proceeds in discrete steps such as cycles, iterations, instruction-executions, task-executions, etc., it is possible, though improbable, that more than one unit may fail in a single step. In this case, an analysis which assumes that at most one active unit can fail at a time is an approximation to the real problem.

Computation of the hardware reliability efficiency index

Mathur and Avizienis (op cit) assume that malfunctions occur according to an exponential failure law. A consequence of this assumption is that at most one unit



Markov diagram of a hybrid configuration

Figure 5

can fail at a given instant which in turn implies that the majority is *always* right. Now consider what happens if the improbable event does occur and the majority is in error and the minority is correct. The correct minority unit will be switched out to be replaced by a spare which is powered up and initialized. A comparison with the other two active units will show that the powered-on spare is in the minority, and it will in turn be switched out to be replaced by yet another spare and so on. Eventually all the spares will be used up and the system will crash. Thus even though the probability of failure of two units in one iteration is indeed small, the consequence of this improbable event is catastrophic. Hence we feel that in calculating SRE it is important to back-up the Mathur-Avizienis study of this ingenious method with an analysis that does not assume that simultaneous failures never occur.

In this analysis we will assume that computation proceeds in discrete steps called tasks; a task may consist of several instructions or a single instruction. Key variables of the active units are compared at the end of a task completion, and the minority element, if any, is switched out. Let the probability of failure of a unit on any step of the computation be P, independent of other units and earlier events. A discrete-state, discrete-transistion Markov process may be used to model this system. A Markov state diagram is shown in Figure 5. If the system is in state F, then a system failure has already occurred. The reliability of the system is the probability that the system is not in state F at the Nth iteration, where N is the number of computation steps required in the mission. The reliability can be computed analytically from the Jordan normal form. A curve of reliability as a function of N is shown in Figure 8. Let R_H be the reliability of the hybrid system, C the cost of each unit and D the cost of the vote-taker. The hardware reliability efficiency index with two spares is then:

$$\text{HRE} = \frac{R_H - (1-P)^N}{4C+D}$$

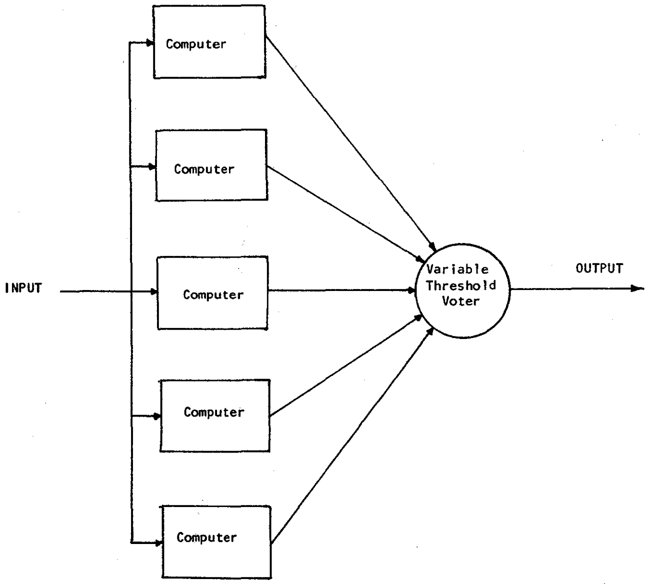
Self-purging system

Discussion

Consider a self-purging system shown in Figure 6. Initially there are five active units and no spares. If at any instant the vote-taker detects a disagreement among the active units, the units whose outputs are in the minority are switched out, leaving three, active, error-free units. If the failure rates for active and passive units are the same, the self-purging system will tolerate two simultaneous failures, which may be catastrophic for the hybrid system.

Computation of the hardware reliability efficiency index

In this analysis we shall assume that computation proceeds in discrete steps, as in the analysis for the



Self-purging System with 5 Units

Figure 6

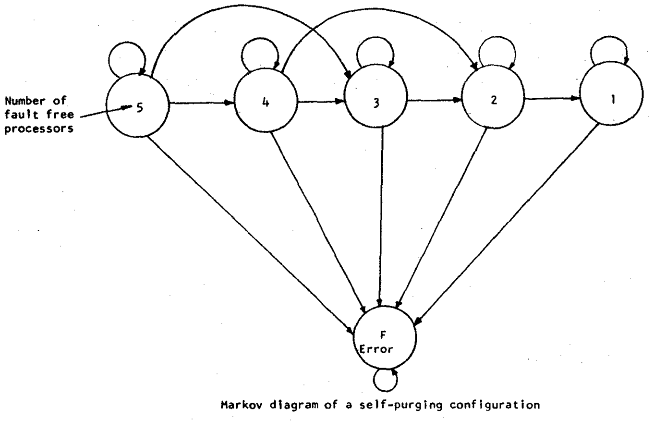

Number of fault free processors
5
4
3
2
1
F
Error
Markov diagram of a self-purging configuration

Figure 7

hybrid system. Let P be the probability of failure of a unit on a computation step, independent of other units and earlier steps. A Markov state diagram for this process is shown in Figure 7. As in the hybrid case the reliability of the system is the probability that the system is not in state F one the Nth computation step. A curve showing the reliability of this system as a function of N is shown in Figure 8. Let R_S be the reliability of a self-purging system with five active units initially. Then

$$\text{HRE} = \frac{R_S - (1-P)^N}{4C+D}$$

If the cost of power supplies are included HRE for the hybrid system is larger than that for self-purging.

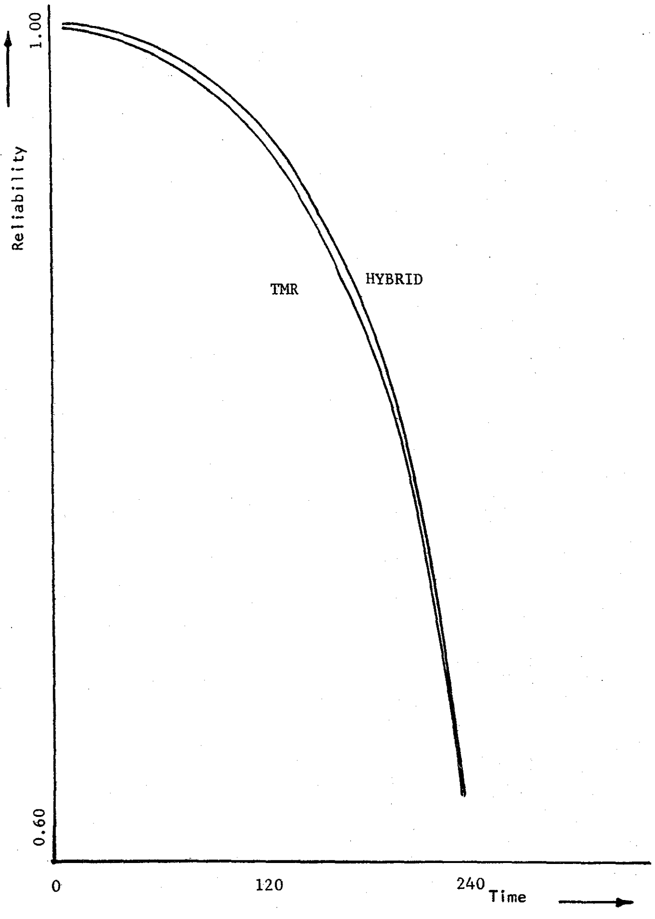


Figure 8

Summary of hardware methods

TMR, hybrid, and a system called a self-purging system were discussed. Some of the problems of approximating these systems as continuous transition systems were analyzed. Techniques for obtaining the hardware reliability efficiency indices were presented. Similar techniques can be used for other hardware methods.

CONCLUSION

We have attempted to develop a set of simple indices which may be useful in comparing different techniques for achieving reliability. We feel that an important research and pedogogical problem is to develop a more comprehensive, sophisticated framework. Models for rollback and discrete transition models for hybrid and self-purging systems were discussed briefly.

ACKNOWLEDGMENT

This research was supported in part by NSF grants GJ-35109 and GJ-492.

REFERENCES

1 J VON NEUMANN
Probabilistic logics and the synthesis of reliable organisms from unreliable components
Automata Studies p 43-98 Princeton University Press Princeton N J 1956

2 F P MATHUR A AVIZIENIS
Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple module redundancy with self-repair
Proc Spring Joint Computer Conference 1970

3 M BALL F H HARDIE
Redundancy for better maintenance of computer systems
Computer Design pp 50-52 January 1969

4 M BALL F H HARDIE
Self-repair in a TMR computer
Computer Design pp 54-57 February 1969

5 A COWAN
Hardware-software tradeoffs in the design of reliable computers
Master's thesis in the Department of Computer Sciences University of Texas December 1971

6 A N HABERMANN
Prevention of system deadlocks
Comm ACM Vol 12 No 7 July 1969

7 J HOWARD
The coordination of multiple processes in computer operating systems
Dissertation Computer Sciences Department University of Texas at Austin 1970

8 K M CHANDY C V RAMAMOORTHY
Optimal rollback
IEEE-C Vol C-21 No 6 pp 546-556 June 1972

9 G OPPENHEIMER K P CLANCY
Considerations of software protection and recovery from hardware failures
Proc FJCC 1968 AFIPS pp 29-37

10 A N HIGGINS
Error recovery through programming
Proc FJCC 1968 AFIPS pp 39-43

11 A N HABERMANN
On the harmonious cooperation of abstract machines
Thesis Mathematics Department Technological
U Eindhoven The Netherlands 1967

Automation of reliability evaluation procedures through CARE—The computer-aided reliability estimation program*

by FRANCIS P. MATHUR

University of Missouri
Columbia, Missouri

INTRODUCTION

The large number of different redundancy schemes available to the designer of fault-tolerant systems, the number of parameters pertaining to each scheme, and the large range of possible variations in each parameter seek automated procedures that would enable the designer to rapidly model, simulate and analyze preliminary designs and through man-machine symbiosis arrive at optimal and balanced fault-tolerant systems under the constraints of the prospective application.

Such an automated procedural tool which can model self-repair and fault-tolerant organizations, compute reliability theoretic functions, perform sensitivity analysis, compare competitive systems with respect to various measures and facilitate report preparation by generating tables and graphs is implemented in the form of an on-line interactive computer program called CARE (for Computer-Aided Reliability Estimation). Essentially CARE consists of a repository of mathematical equations defining the various basic redundancy schemes. These equations, under program control, are then interrelated to generate the desired mathematical model to fit the architecture of the system under evaluation. The math model is then supplied with ground instances of its variables and then evaluated to generate values for the reliability theoretic functions applied to the model.

The math models may be evaluated as a function of absolute mission time, normalized mission time, non-redundant system reliability, or any other system parameter that may be applicable.

Unifying notation

A unifying notation, developed to describe the various system configurations using selective, massive, or hybrid redundancy is illustrated in Figure 1.

N refers to the number of replicas that are made massively redundant (NMR); S is the number of spare units; W refers to the number of cascaded units, i.e., the degree of partitioning; R( ) refers to the reliability of the system as characterized in the parentheses; TMR stands for triple modular redundant system (N=3); the NMR stand for N-tuple modular redundancy.

A hybrid redundant system H(N, S, W) is said to have a reliability R(N, S, W). If the number of spares is S=0, then the hybrid system reduces to a cascaded NMR system whose reliability expression is denoted by R(N, O, W); in the case where there are no cascades, it reduces to R(N, O, 1), or more simply to R(NMR). Thus the term W may be elided if W=1. The sparing system R(1, S) consists of one basic unit with S spares.

Furthermore, the convention is used that R* indicates that the unreliability $(1-R_v)$ due to the overhead required for restoration, detection, or switching has been taken into account e.g., $R^*(NMR)=R_v.R(NMR)$; if the asterisk is elided then it is assumed that the overhead has a negligible probability of failure. This proposed notation is extendable and can incorporate a number of functional parameters in addition to those shown here by enlarging the vector or lists of parameters within the parentheses, e.g., R(N, S, W, . . . , X, Y, Z).

Existing reliability programs

Some reliability evaluation programs, known to the author, are the RCP, the RELAN, and the REL70. The RCP[1,2] is a reliability computation package developed by Chelson (1967). This is a program which

* The work presented here was carried out while the author was with the Jet Propulsion Laboratory, California Institute of Technology, and under Contract No. NAS7-100, sponsored by the National Aeronautics and Space Administration.

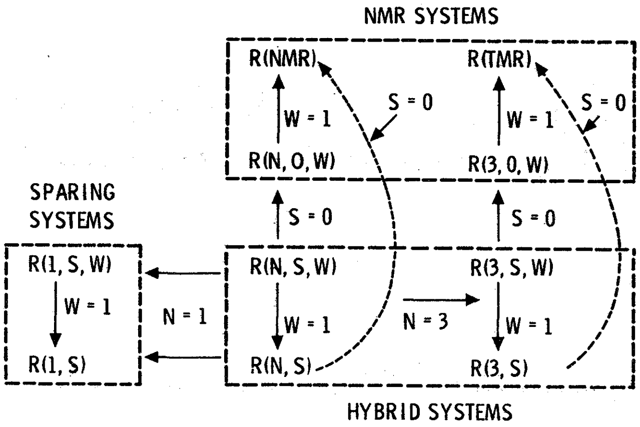


Figure 1—Unifying notation

can model a network of arbitrary series-parallel combinations of building blocks and analyzes the system reliability by means of probabilistic fault-trees. RELAN[3] is an interactive program developed by TIME/WARE and is offered on the Computer Sciences Corporation's INFONET network. RELAN like RCP models arbitrary series-parallel combinations but in addition allows a wide choice (any of 17 types) of failure distributions. RELAN has concise and easy to use input formats and provides elegant outputs such as plots and histograms. REL70[4] and its forerunner REL[5] developed by Bouricius, et al., are interactive programs developed in APL/360. Unlike RCP and RELAN, REL70 is more adapted for evaluating systems other than series-parallel such as standby-replacement and triple modular redundancy. It offers a large number of system parameters, in particular C the coverage factor defined as the probability of recovering from a failure given that the failure exists and Q, the quota, which is the number of modules of the same type required to be operating concurrently. REL70 is primarily oriented toward the exponential distribution though it does provide limited capabilities for evaluating reliability with respect to the Weibull distribution; its outputs are primarily tabular. Since APL is an interpretive language, REL is slow in operation; however, its designers have overcome the speed limitation by not programming the explicit reliability equations but approximate versions[6] which are applicable to short missions by utilizing the approximation $(1-\exp(-\lambda T)) = \lambda T$ for small values of λT.

The CARE program is a general program for evaluating fault-tolerant systems, general in that its reliability theoretic functions do not pertain to any one system or equation but to all equations contained in its repository and also to complex equations which may be formed by interrelating the basic equations. This repository of equations is extendable. Dummy routines are provided wherein new or more general equations may be placed as they are developed and become available to the fault-tolerant computing community. For example, the equation developed by Bouricius, et al., for standby-replacement systems embodying the parameters C and Q has been bodily incorporated into the equation repository of CARE.

CARE'S ENVIRONMENT, USERS AND AVAILABILITY

CARE consists of some 4150 FORTRAN V statements and was developed on the UNIVAC 1108 under EXEC 8 (version 11C). The particular FORTRAN V compiler used was the Level 7E having the modified 2/3/4 argument ENCODE-DECODE commands. The amount of core required by the unsegmented CARE is 64K words. The software for graphical outputs is designed to operate in conjunction with the Stromberg Carlson 4020 plotter. The software enabling three-dimensional projections, namely the Projectograph routines,[7] are a proprietary item of Technology Service Corporation.

In addition to the Jet Propulsion Laboratory, the originator, currently there are three other users of CARE, namely NASA Langley Research Center (a FORTRAN II version operational on a CDC 3600), Ultrasystems Corp. (operational on a UNIVAC 1108 under EXEC II), and MIT Draper Laboratory. The CARE program, minus the Projectograph routines, has been submitted to COSMIC** and is available to interested parties from them along with users manuals. Its reference number at COSMIC is NPO-13086.

CARE's repository of equations

The equations residing in CARE, based on the exponential failure law, model the following basic fault-tolerant organizations:

(1) Hybrid-redundant (N, S) systems.[8,9]
  (a) NMR (N, 0) systems.[10]
  (b) TMR (3, 0) systems.[10]
  (c) Cascaded or partitioned versions of the above systems.
  (d) Series string of the above systems.

The equation representing the above family of

** Computer Software Management and Information Center, University of Georgia, Athens, Georgia 30601.

systems is the following:

$$R^*(N, S)$$

$$= \Bigg[R^{N/W}RS_s^{S/W} \Big[1 + \sum_{j=0}^{S-2} \binom{NK+S}{j+1} \Big(\frac{1}{R_s^{1/W}} - 1 \Big)^{j+1}$$

$$+ \sum_{i=0}^{n} \binom{N}{i} \binom{NK+S}{S} \sum_{l=0}^{i} \frac{\binom{i}{l} (-1)^{i-l}}{\binom{Kl+S}{S}}$$

$$\times \Big\{ \Big(\frac{1}{R_s^{S/W} R^{l/W}} - 1 \Big)$$

$$- \sum_{j=0}^{S-2} \binom{Kl+S}{j+1} \Big(\frac{1}{R_s^{1/W}} - 1 \Big)^{j+1} \Big\} \Big] RV \Bigg]^{WZ}$$

for $1 \le K < \infty$ and $S > 1$

$$= \Big\{ R^{N/W} R_s^{1/W} \Big[1 + (NK+1) \sum_{i=0}^{n} \binom{N}{i} \sum_{l=0}^{i} \binom{i}{l}$$

$$\times \frac{(-1)^{i-l}}{(Kl+1)} \Big(\frac{1}{R_s^{1/W} R^{l/W}} - 1 \Big) \Big] RV \Big\}^{WZ}$$

for $1 \le K < \infty$ and $S = 1$

(2) Standby-sparing redundant (1, S) systems.[6,10]

(a) K-out-of-N systems.[6]
(b) Simplex systems.
(c) Series string and cascaded versions of the above.

The general equation for the above is:

$$R(1, S) = \Big[R^{Q/W} \Big\{ 1 + \sum_{i=1}^{S} \Big[\frac{c^i}{i!} (1 - R_s^{1/W})^i$$

$$\times \prod_{j=0}^{i-1} (QK+j) \Big) \Big\} \Big]^{WZ}$$

for $1 \le K < \infty$

$$= \Big[R^{Q/W} \sum_{i=0}^{S} \frac{(CQ\lambda T/W)^i}{i!} \Big]^{WZ}$$

for $K = \infty$

(3) TMR systems with probabilistic compensating failures.[10]
(a) Series string and cascaded versions of the above.

The equation characterizing this system is:

$$R^*(3, 0) = \{ RV[3R^{2/W} - 2R^{3/W}$$
$$+ 6P(1-P)R^{1/W}(1-R^{1/W})^2] \}^{WZ}$$

(4) Hybrid/simplex redundant $(3, S)_{sim}$ systems.[11]
(a) TMR/simplex systems.[6]
(b) Series string and cascaded versions of the above.

The general equation for this class of systems is the following:

$$R(3, S)_{sim}[T]$$

$$= R^3 R_s^S \Big\{ 1 + 1 \cdot 5 \Big(\frac{1}{R^2 R_s^S} - 1 \Big) \prod_{i=1}^{S} \Big(\frac{3K+i}{2K+i} \Big)$$

$$- \prod_{j=1}^{S} \frac{(3K+j)}{j} \sum_{i=0}^{S-1} \binom{S}{i} (-1)^i$$

$$\times \Big(\frac{1}{R_s^{S-1}} - 1 \Big) \frac{3K^2}{(2K+i)(3K+i)} \Big\}$$

for $S > 0$ and $\mu > 0$

and

$$= (1 \cdot 5)^{S+1} R - R^3 \sum_{i=1}^{S} \frac{(3\lambda T)^{S+1-i}}{(S-i)!}$$
$$\times [(1 \cdot 5)^i - 1] - R^3[(1 \cdot 5)^{S+1} - 1]$$

for $S > 0$ and $\mu = 0$

and

$$R^*(3, S)_{sim} = R_v \cdot R(3, S)_{sim}$$

For the description of the above systems and their mathematical derivations, refer to the cited references. These equations are the most general representation of their systems parameterizing mission time, failure rates, dormancy factors, coverage, number of spares, number of multiplexed units, number of cascaded units, and number of identical systems in series. The definitions of these parameters reside in CARE and may be optionally requested by the user. More complex systems may be modeled by taking any of the above listed systems in series reliability with one another.

TABLE I—Table of Abbreviations and Terms

λ	=Powered failure rate
μ	=Unpowered failure rate
K	$=\lambda/\mu$ = Dormancy factor
T	=Mission time
λT	=Normalized mission time
R	=Simplex reliability
R	=Dormant reliability, $\exp(-\mu T)$.
S	=Number of spares
n	=(N-1)/2 where N is the total number of multiplexed units
Q	=Quota or number of identical units in simplex systems
C	=Coverage factor, Pr(recovery/failure)
RV	=Reliability of restoring organ or switching overhead
Z	=Number of identical systems in series
W	=Number of cascaded or partitioned units
P	=Probability of unit failing to "zero"
TMR	=Triple modular redundancy
TMR_p	=TMR system with probabilistic compensating failures
(1,S)	=Standby spare system
(N,S)	=Hybrid redundant system
$(3,S)_{sim}$	=Hybrid/simplex redundant system
MTF	=Mean life
R(MTF)	=Reliability at the mean life

Reliability theoretic functions

The reliability equations in the repository may be evaluated as a function of absolute mission time (T), normalized mission time (λT), nonredundant system reliability (R), or any other system parameter that may be applicable. The set of reliability theoretic functions defined in CARE are applicable to *any* of the equations in the repository. This independence of the equations from the functions to be applied to the equations impart generality to the program. Thus the equation repository may be upgraded without effecting the repertoire of functions. The various reliability theoretic functions useful in the evaluation of fault-tolerant computing systems have been presented in Ref. 11, the measures of reliability have been defined, categorized into the domains of probabilistic measures and time measures and their effectiveness compared. Among the various measures of reliability that the user may request for computation are: the system mean-life, the reliability at the mean-life, gain in reliability over a simplex system or some other competitive system, the reliability improvement factor, and the mission time availability for some minimum tolerable mission reliability.

Operational features

Although CARE is primarily an interactive program, it may be run in batch mode if the user prespecifies the protocol explicitly. In the interactive mode CARE assumes minimum knowledge on the user's part. Default values are provided to many of the items that a user should normally supply. This safeguards the user and also makes usage simpler by providing logical default values to conventionally used parameters. Instructions provided by CARE are optional thus the experienced user can circumvent these and operate in fast mode. Definitions of reliability terms and abbreviations used in the program may be optionally requested. An optional "echo" feature that echoes user's responses back to the terminal is also provided. A number of diagnostics and recovery features that save users from some common fatal errors are in the program.

Model formulation—an example

A typical problem submitted for CARE analysis may be the following: Given a simplex system with 8 equal modules which is made fault-tolerant by providing two standby spares for each module, where each module has a constant failure rate of 0.5 failures per year and where the spares have a dormancy factor of 10 and the applicable coverage factor being 0.99, it is required to evaluate the system survival probability in steps of 1/10 of a year for a maximum mission duration of 12 years. It is required that the system reliability be compared against the simplex or nonredundant system and that all these results be tabulated and also plotted. It is further required that the mean-life of the system às well as the reliability at the mean-life be computed. It is of interest to know the maximum mission duration that is possible while sustaining some fixed system reliability objective and to display the sensitivity of this mission duration with respect to variations in the tolerable mission reliability.

It is also required that the above analysis be carried out for the case where three standby spares are provided and these configurations of three and two spares be compared and the various comparative measures of reliability be evaluated and displayed.

The above problem formulation is entered into CARE by stating that Equation 2 (which models standby spare systems) is required and the pertinent data (S=2,3; Z=8; K=10; T=12.0; LAMBDA=0.5; C=0.99; STEP=0.1) is inserted into CARE between the VARiable namelist delimiters $VAR . . . $END.

The above example illustrates the complexity of problems that may be posed to CARE, and the simplicity with which the specifications are entered. The reliability theoretic functions to be performed on the above specified system are acknowledged interactively by responding a YES or NO on the demand terminal to CARE's questions at the time it so requests.

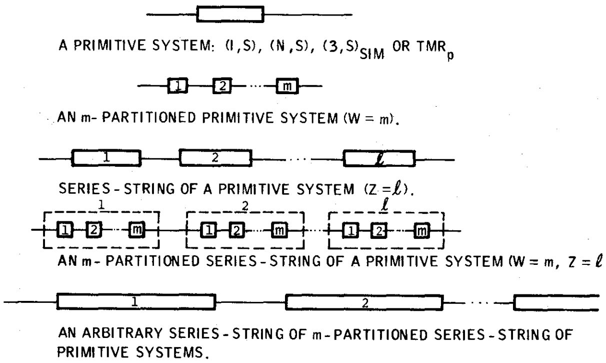

A PRIMITIVE SYSTEM: (1,S), (N,S), (3,S)SIM OR TMRp
1
2
m
AN m- PARTITIONED PRIMITIVE SYSTEM (W = m).
1
2
ℓ
SERIES - STRING OF A PRIMITIVE SYSTEM (Z = ℓ).
1
2
ℓ
AN m- PARTITIONED SERIES - STRING OF A PRIMITIVE SYSTEM (W = m, Z = ℓ).
1
2
AN ARBITRARY SERIES - STRING OF m - PARTITIONED SERIES - STRING OF PRIMITIVE SYSTEMS.

Figure 2—Formation of complex systems

COMPLEX SYSTEMS

The basic equations in CARE's repository define the primitive systems: (1, S), (N, S), $(3, S)_{\text{sim}}$ and TMR_p. Equations representing more complex systems may be fabricated by combining the primitive systems in series reliability with one another as shown in Figure 2.

The description of a complex system is entered by first enumerating the equation numbers of the primitive systems involved in namelist VARiable1. Thus "$VAR1; PROD=1, 2; $END;" states that equation 1 and equation 2 are to be configured in series reliability. Next, the parameter specifications for these equations are then entered using the namelist VARiable.

The set of values for any parameter pertaining to a complex system is stored as a matrix, thus in the general case of PARAMETER (m, n) n refers to the equation involved m is the internal index for the set of values that will be attempted successively. For example, C(1, 2) = 1.0, 0.99 states that in equation 2 (the equation for standby-spares system) the value of the coverage factor should be taken to be 1.0 and having evaluated the complex system for this value the system is to be reconsidered with coverage factor being 0.99.

Complex model formulation—an example

It was required to evaluate a system consisting of 8 equally partitioned modules in a standby-spares (1, S) configuration having 2 spares for each module. The 9th module was the hard-core of the system and was configured in a Hybrid redundant (3, S) system having 2 spares (S=2). The coverage on the (1, S) system modules was to be initially considered to be 1.0. The lower bound on the failure rate λ on all the modules had been evaluated to be .01752 failures/year on the basis of parts count. This complex system as specified here was to be evaluated for the worst case dormancy factors K of 1 and infinity.

On completing the evaluation of the above system, the effect of reducing coverage to 0.99 was to be reevaluated. Also the effect of increasing the number of spares to 3, as also the effect of increasing the module failure rates to their upper bound value of .0876 failures/year. All combinations of these modifications on the original system are to be considered. The mission time is 12 years and evaluations are to be made in steps of 1/10th of a year.

The above desired computations are specified using the VAR namelist thus:

```
$VAR; T=12.0; STEP=0.1; Z(1,1)=1,
Z(1,2)=8; C(1,2)=1.0, 0.99; N(1,1)=3;
S(1,1)=2,3,S(1,2)=2,3;LAMBDA(1,1)=
.01752, .0876, LAMBDA(1, 2)=.01752,
.0876;K(1,1)=1.0, INF, K(1,2)=1.0, INF;
$END;
```

(Note the semicolons (;) denote carriage returns.) The ease and compactness with which complex systems can be specified in CARE is demonstrated by the above example. The reader will note the complex system configured in this example corresponds to a STAR-like system having eight functional units in standby-spare mode and a hard-core test-and-repair unit in Hybrid redundant mode (Figure 3).

SOME SIGNIFICANT RESULTS USING CARE

Some significant results pertaining to the behavior of W partitioned NMR system (Figure 4) will now be presented. These results pertain to the behavior or reliability theoretic functions of an NMR system such as its mean life or mean time to first failure (MTF) and the reliability of the system at the mean life, R(MTF). The reliability theoretic system measure—

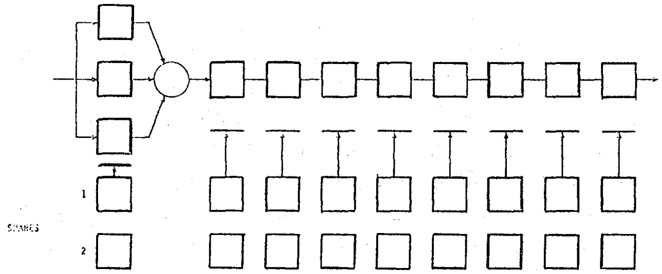


Figure 3—Configuration for an example of a STAR-like complex model

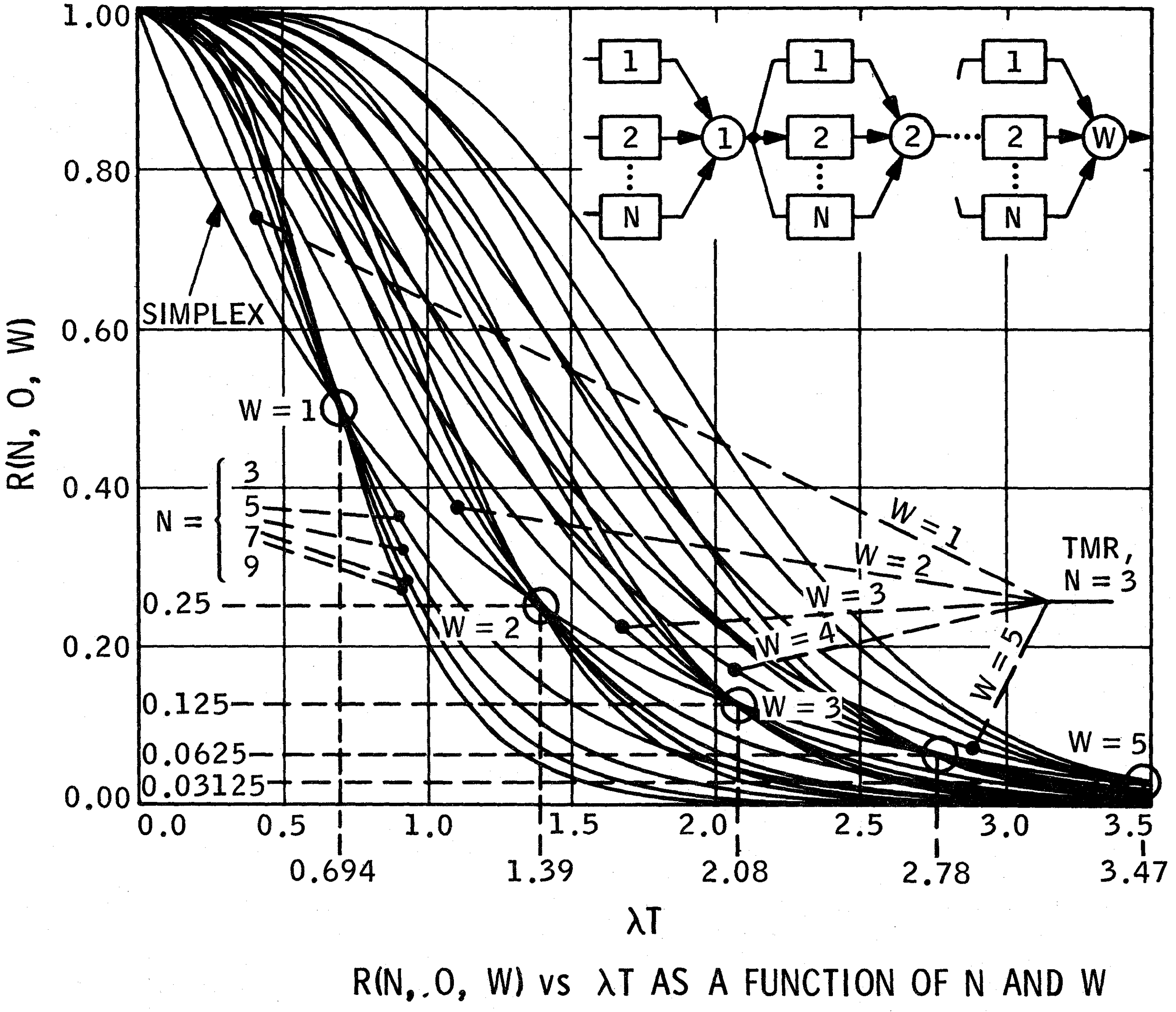


Figure 4—R(N,0,W) vs λT as a function of N and W

reliability at the mean life, R(MTF)—is the reliability of the system computed for missions or time durations of length equal to the mean time to first failure of the system. The behavior of these functions were evaluated under the limiting conditions of the system parameters in order to establish system performance bounds. The results presented here have been both proven mathematically[10] and been verified by CARE analysis.

Since it is well-known that mean-life (MTF) is not a credible measure of reliability performance (e.g., MTF of a simplex system is greater than the MTF of a TMR system!), another measure the reliability at the mean-life R(MTF) has been used to supplant MTF. This measure essentially uses a typical reliability estimate of the system. The typical reliability value being taken at a specific and hopefully a representative time of the system. This representative time is taken to be the time to first failure of the system, namely the MTF of the system. The foregoing is the rationale for choosing R(MTF) as a viable measure of system reliability.

However, contrary to general belief this measure R(MTF) is not a good measure for partitioned NMR systems due to its asymptotic behavior as a function of the number of partitions W. It is proved in [10] that the reliability at MTF of a (3, O, W) system in the limit as W becomes very large approaches the value exp $(-\pi/4)$ asymptotically from below and that this bounding value is reached very rapidly, see Figure 5.

TABLE II—MTF and R(MTF) as a Function of W

(3,0,W) System		
W	MTF	R(MTF)
0 (Simplex)	1.0	0.368
1 (TMR)	0.83	0.402
∞(3,0,∞)	∞	$\exp(-\pi/4) \approx 0.454$

Some other results observed graphically in Figure 4 and the detailed mathematical proof of which are in [10] are summarized below. These results follow from the general reliability equation for a W partitioned NMR system, which is:

$$R(N, O, W) = \left[\sum_{i=0}^{(N-1)/2} \binom{N}{i} \cdot (1-R^{1/W})^{i} \cdot R^{(N-i)/W} \right]^{W}$$

and that the normalized mean-life,

$$MTF(N, O, W) = \int_0^\infty R(N, O, W)\, d\lambda t$$

The bounds on the mean-life as a function of the degree of redundancy N is:

$\lim_{N\to\infty} MTF(N, O, W) = W \ln 2$ where ln is the Naperian logarithm.

and in the particular case of W = 1

$$\lim_{N\to\infty} MTF(NMR) = \ln 2 \approx 0.694$$

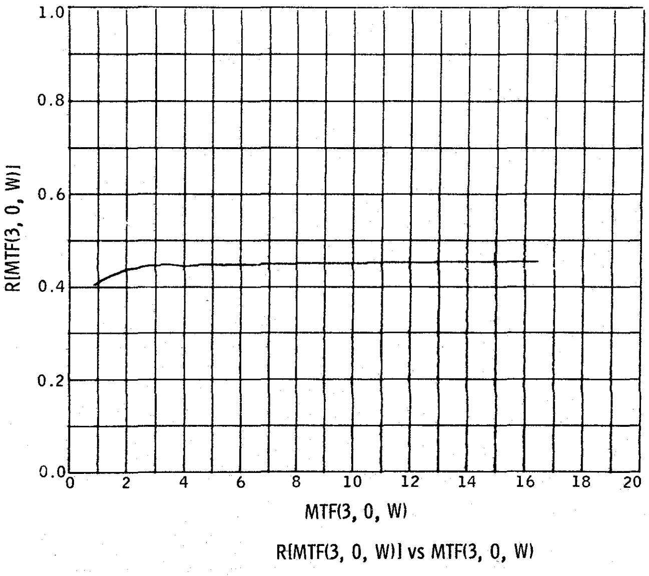


Figure 5—R[MTF(3,0,W)] vs MTF (3,0,W)

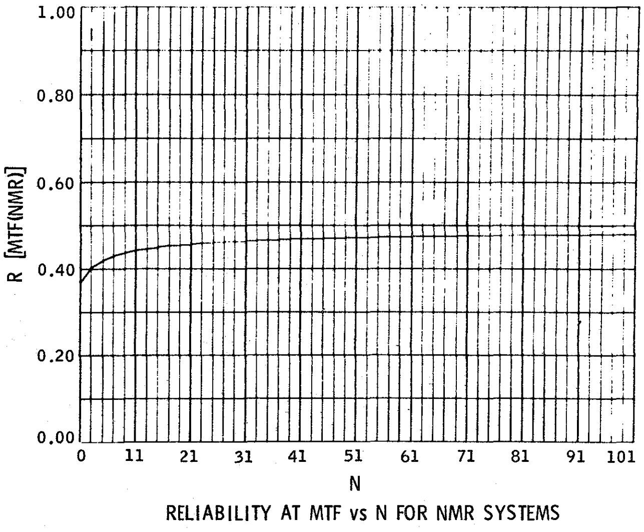


Figure 6—Reliability at MTF vs N for NMR systems

Also, for the reliability at the MTF:

$\lim_{W\to\infty} R(MTF)$ of $(3, O, W) = exp(-\pi/4) \approx 0.454$

and $\lim_{N\to\infty} R(MTF)$ of $NMR = 0.5$

The family of reliability curves representing the NMR system as shown in Figure 4 exhibits the classical cross-over point which for (3, 0, 1) system occurs at the coordinates $R_{sys} = 0.5$ and $\lambda T = 0.694$. The general specification of the coordinates of the cross-over point for arbitrary values of N and W may be expressed as follows:

Cross-over point $[R; \lambda T]$ of a (N, O, W) system $= [(0.5)^W; \lim_{N\to\infty} MTF(N, O, W)]$

$$= [(0.5)^W; W \ln 2]$$

These results are tabulated in Tables II and III.

TABLE III—Coordinates of Cross-Over Point [R; λT] as a Function of W and N

(N,0,W) System	
W	[R; λT]
1	0.5; ln 2
2	$(0.5)^2$; 2 ln 2
x	$(0.5)^x$; x ln 2

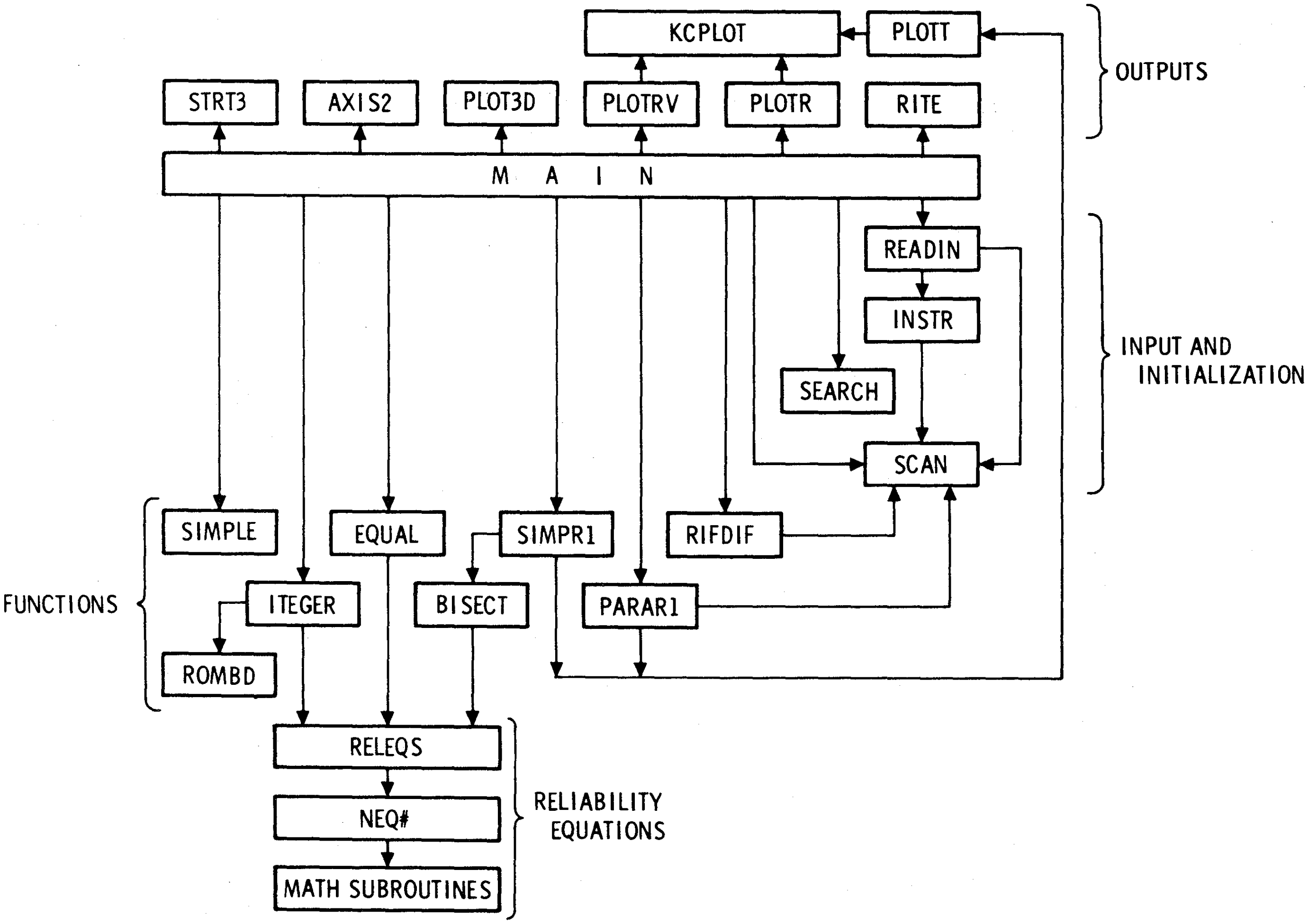


Figure 7—CARE's structure

CARE'S STRUCTURE

The foregoing described the performance capabilities of CARE; in this section the implementation structure is described.

CARE consists of a number of primary subroutines. The relationship amongst these primary subroutines is shown in the simplified flow diagram of Figure 7.

The overall program has four broadly defined segments:

(i) dealing with reading in of data and initializing of the logical flow of the program,
(ii) dealing with the functions that are to be performed using the input data,
(iii) dealing with the repository of the general equations that model fault-tolerant systems and the relevant mathematical routines required to evaluate these equations, and
(iv) dealing with initializing output formats, passing the data, and outputting it as 2D plots, 3D projections, or as tables.

All these four segments are under the control of MAIN which sets the DO loops and determines what and how many times each function is to be performed and controls the mode in which the results are to be outputted.

Parameter Handling

The system parameters, LAMBDA, Mu, S, N, K, Q, C, RV, Z, W, and P are two dimensional parameter arrays, dimensioned as being 16×NPT and reside in the labeled COMMON/PARA/. Sixteen is the maximum number of values that any one parameter may be assigned in $VAR. The NPT (short for "number of products") pertains to the total number of equations

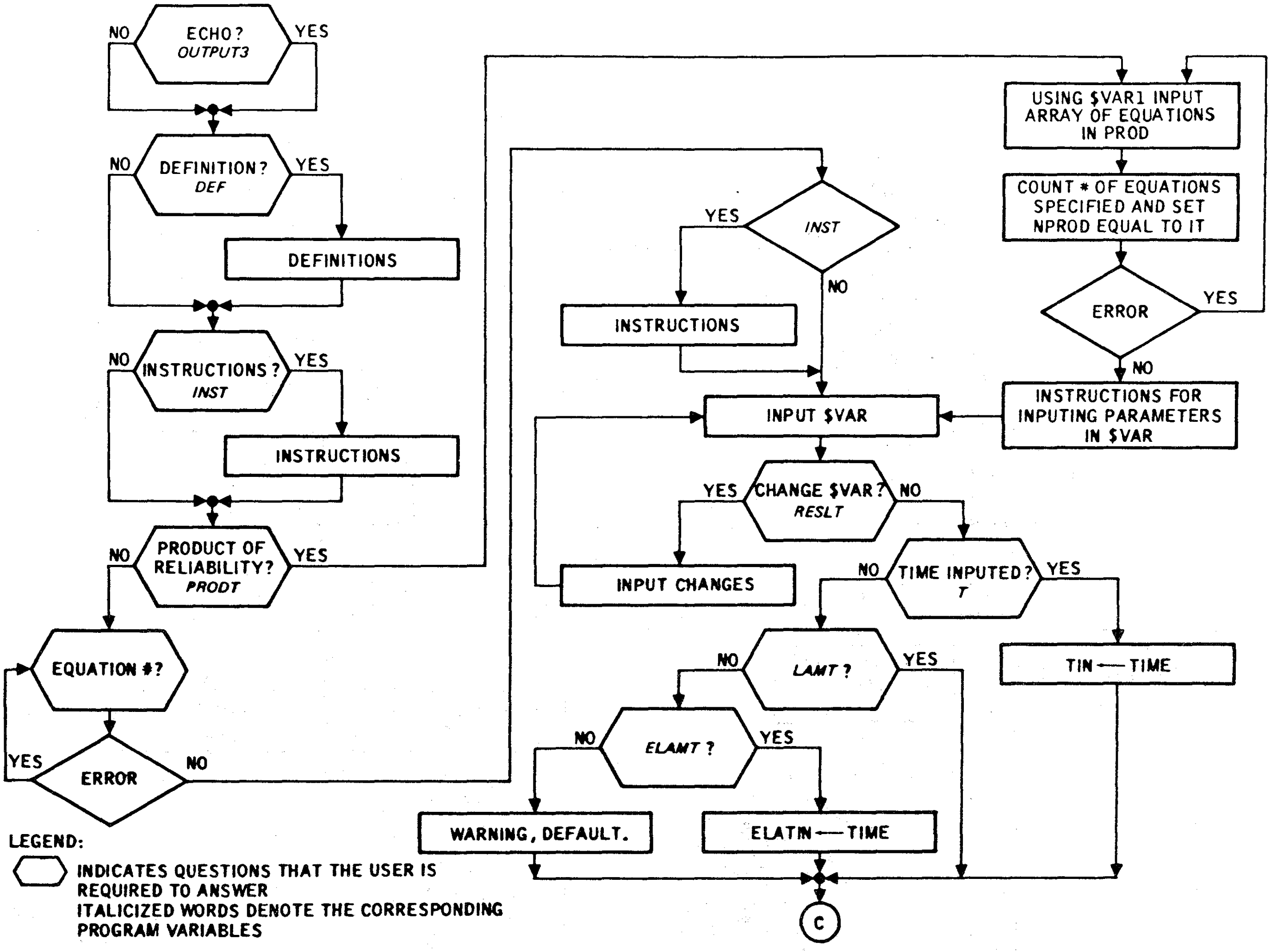


Figure 8—Flowchart of CARE's protocol

that may be used in forming the product. If a complex equation is not being formed, then NPT=1. The maximum value that NPT can currently take is 10. Thus the rows of the parameter matrices contain the values of the parameter while the columns contain the index of the equation numbers (with reference to the order in which they were entered in $VAR1) that these parameters pertain to.

The time pertinent parameters, such as Time, LAMT (λT) and ELAMT (exp($-\lambda$T)) are single valued. Their values are the maximum values that the parameter is to take, the incremental steps at which computations are to be performed is specified by assigning a value to the variable STEP in $VAR (the default value for STEP is one).

The *number of values* specified for each parameter is determined by the subroutine SEARCH, these values then form the values of the DO limits in the MAIN program. The *actual value* is obtained by accessing the particular element of the 16×NPT parameter matrix.

Logical relationship between the routines

As shown in Fig. 7, MAIN is the driver for the CARE program. MAIN calls READIN, the subroutine READIN writes out questions for the user to answer and records his answers. These questions are asked in a logical manner with a large number of options permitting the user flexibility in the specification of his problem. A large number of diagnostics and automatic recovery from user's input errors are provided.

Typically, READIN writes out a question, reads in the user's answer to the question, and if the echo feature had been requested, READIN echoes back the

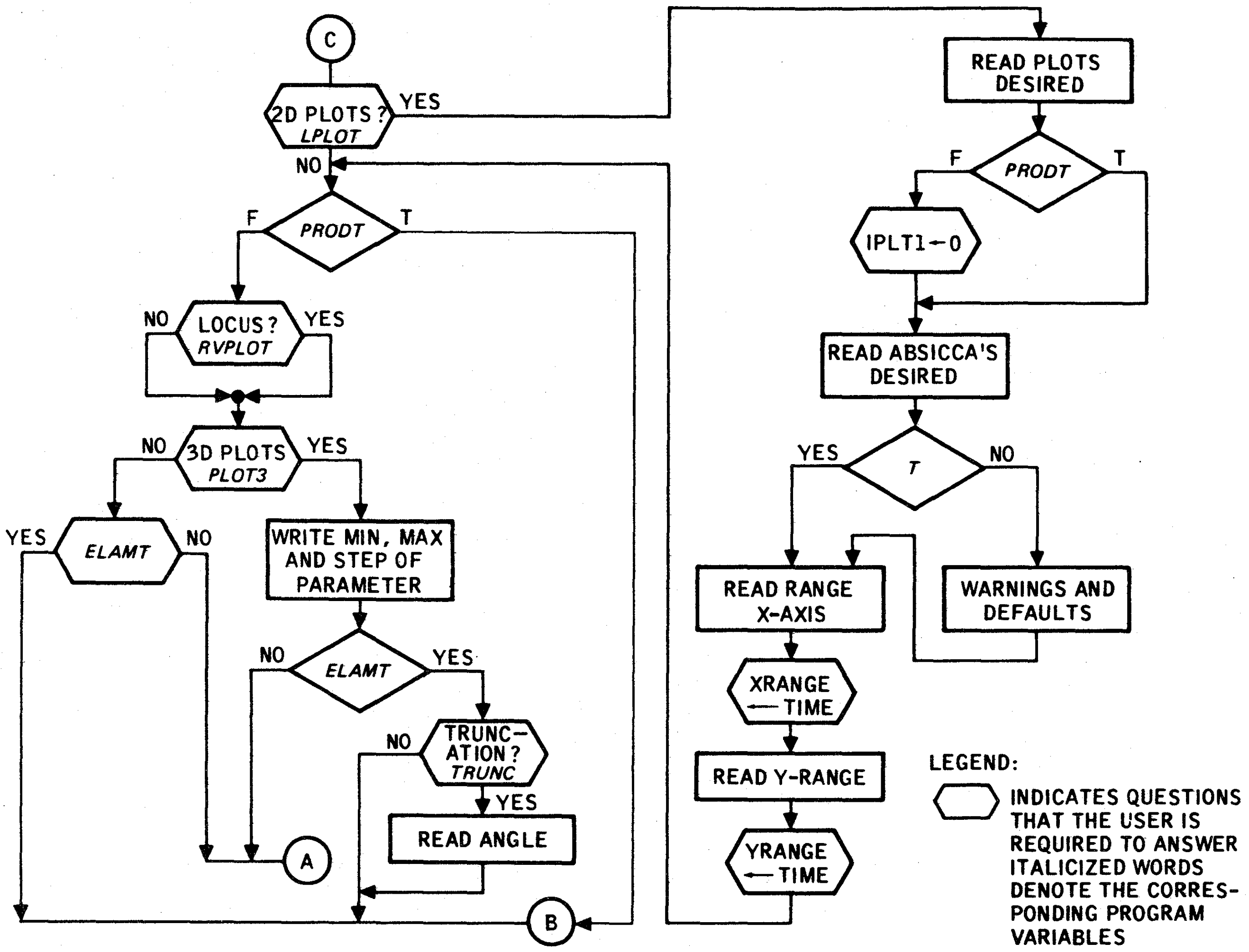


Figure 8 (continued)

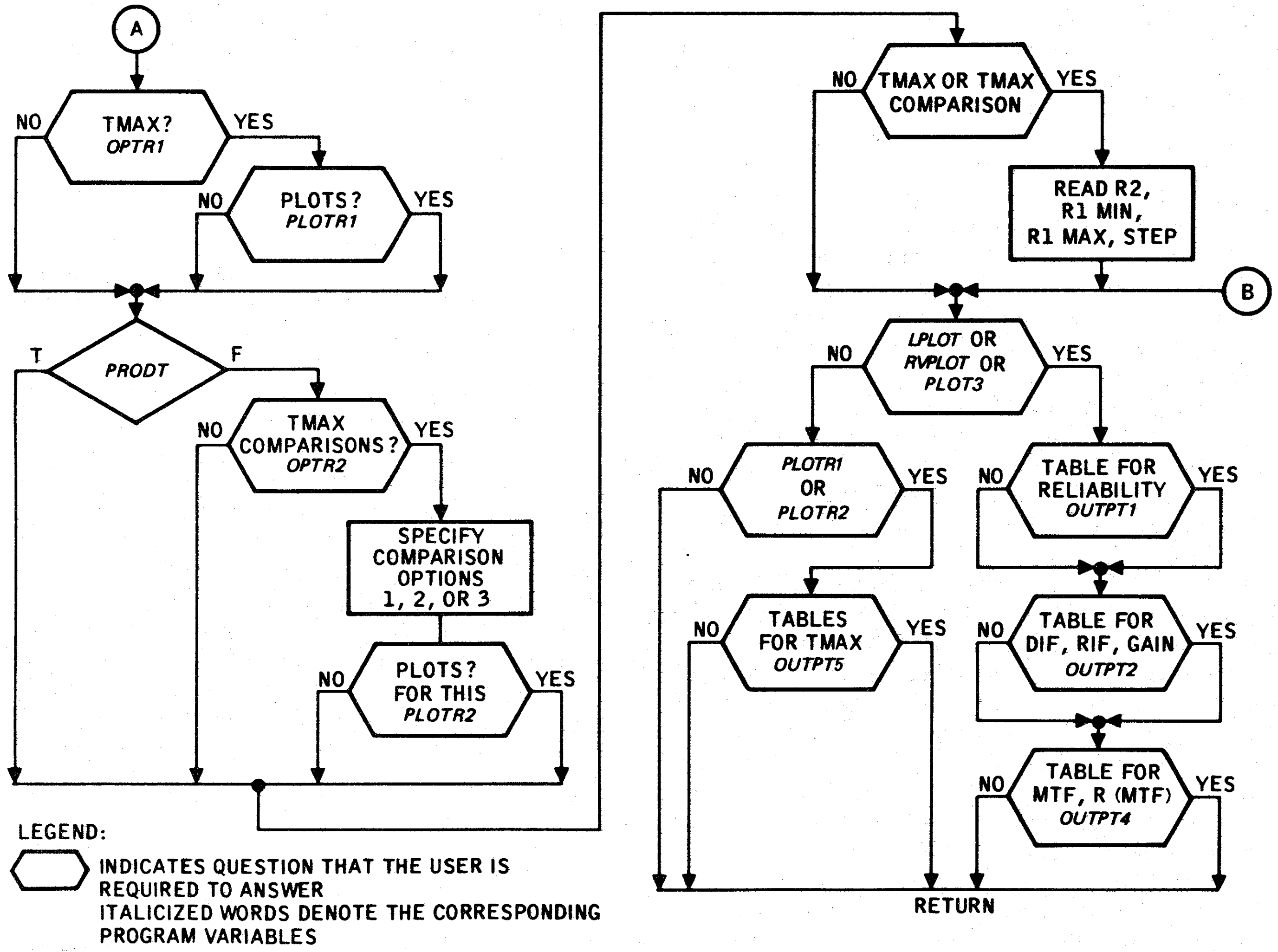


Figure 8 (continued)

answer just read. READIN then calls SCAN passing to it the array containing the information read-in for recognition. SCAN determines whether the answer was a YES or a NO or whether it was a parameter input; if it was a parameter input, then it determines its identity. If an input error is detected, the user is asked to try again. This is implemented by using the $RETURN call parameter feature which returns from SCAN jumping back to preceding read of answer statement in the calling program (READIN). READIN thus gathers input data from the user thus determining which subroutines and features need to be called and in what order. The logic of READIN and the decision tree that the user has to traverse is shown in Figure 8.

Returning from READIN, MAIN calls SEARCH. SEARCH proceeds to count the number of values that were inputted for each of the system parameters, these determine how many times a particular subroutine or function has to be iterated.

Returning from SEARCH, MAIN asks the user to specify which parameter shall be the family variable, the user's response is read, optionally echoed back and recognized by SCAN. MAIN then determines which one of three possible time parameters—T, λT, or exp($-\lambda$T)—had been inputted. MAIN then prepares the DO loop limits and rearranges their order in accordance with the inputted family parameter. The inherent nested order of the DO loops with respect to the system parameters is LAMBDA, Mu, S, N, K, Q, C, RV, Z, W, and P. This initial ordering of the parameter requires to be changed since (i) any of these parameters may be specified to be the family parameter and (ii) since the innermost DO loop must necessarily correspond to this family parameter. Thus effectively the original position of the parameter selected is interchanged with the innermost parameter, namely P.

MAIN also requires to call the subroutine RELATE in order to determine the unspecified parameters of the class λ, μ, λT, μT, exp($-\lambda$T) and K. Since these parameters are interrelated, hence not all may have been directly inputted. RELATE determines values for those parameters unspecified by knowing the ones that were explicitly inputted.

MAIN, using the subroutine RITE, writes the table header for the table of reliability calculations. The header identifies the equation number and the parameters involved. MAIN then calls RELEQS which in turn supplies the desired reliability equation with the necessary parameter values in order to perform the basic reliability calculation. The respective equation subroutines make use of the standard FORTRAN math routines and the math routines provided by CARE.

Depending on the options read-in by READIN, MAIN then calls upon the subroutines that evaluate the functions to be performed such as MTF and reliability at MTF by subroutine INTEGER, differences and gain in reliabilities by subroutine RIFDIF, etc. Finally, MAIN asks if the user wishes to specify another parameter as the family parameter in which case the date read-in by READIN is retained and using the new family parameter MAIN starts its new cycle.

CONCLUSION

A significant portion of concepts and techniques of fault-tolerant computing is embodied in the implementation of this Computer-Aided Reliability Estimation program. Both the performance capabilities and implementation structure have been described here.

The advantages offered by such a special purpose procedural program are that (i) it is conversational, fast and easy to use, (ii) no other program exists that implements CARE functions, (iii) CARE is general in that all its functions pertain to all equations, (iv) has the ability to form complex equations from primitives, (v) the equation repository is extendible, and (vi) has efficient input-output and data handling.

The need and usefulness of such a program to the fault-tolerant computing community is evidenced by the growing number of users of CARE. It is hoped that this description of CARE will motivate and aid practitioners to write more powerful reliability evaluation programs.

ACKNOWLEDGMENTS

The programming support given by G. L. Winje of the Data Systems Division of JPL is gratefully acknowledged. The author also wishes to thank A. Avizienis, D. Rennels, J. Rohr, D. Rubin, and J. Wedel of the Astrionics Division of JPL for the benefit of useful discussions on the subject of fault-tolerant computing.

REFERENCES

1 P O CHELSON
Reliability math modeling using the digital computer
Jet Propulsion Laboratory TR-32-1089 April 1967

2 P O CHELSON
Reliability computation using fault tree analysis
Jet Propulsion Laboratory TR-32-1542 December 1971

3 COMPUTER SCIENCES CORPORATION
RELAN: Reliability analysis package
CSC Sales Brochure No 333 1970

4 W C CARTER et al
Design techniques for modular architecture for reliable computer systems
IBM T J Watson Research Center Report No 70-208-0002 March 1970

5 W G BOURICIUS W C CARTER J P ROTH P R SCHNEIDER
Investigations in the design of an automatically repaired computer
Digest of the First Annual IEEE Computer Conference Sept 1967 pp 64-67

6 J P ROTH W G BOURICIUS W C CARTER P R SCHNEIDER
Phase II of an architectural study for a self-repairing computer
IBM Report SAMSO TR-67-106 Nov 1967

7 TECHNOLOGY SERVICE CORPORATION
Projectograph user's manual
Santa Monica Calif Sept 1969

8 F P MATHUR
Reliability modeling and analysis of a dynamic TMR system utilizing standby spares
Proceedings of the Seventh Annual Allerton Conference on Circuit and System Theory University of Illinois Urbana October 8-10 1969 pp 243-252

9 F P MATHUR A AVIZIENIS
Reliability analysis and architecture of a hybrid redundant digital system: generalized triple modular redundancy with self-repair
AFIPS Conference Proceedings (Spring Joint Computer Conference) Vol 36 Atlantic City May 5-7 1970

10 F P MATHUR
Reliability modeling and architecture of ultra-reliable fault-tolerant digital computers
PhD Thesis University of California at Los Angeles Computer Sciences Dept June 1970 University Microfilms Inc Ann Arbor Mich Reorder No 71-662

11 F P MATHUR
On reliability modeling and analysis of ultrareliable fault-tolerant digital computers
Special Issue on Fault-Tolerant Computing IEEE Transaction on Computers Vol C-20 No 11 Nov 1971 pp 1376-1382

12 A AVIZIENIS G C GILLEY F P MATHUR D A RENNELS J A ROHR D K RUBIN
The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design
Special Issue on Fault-Tolerant Computing IEEE Transaction on computers Vol C-20 No 11 Nov 1971 pp 1312-1321

APPENDIX

CARE's protocol and sample run

```
XQT ATMAN.CARE
HELLO TERMINAL—I AM YOUR RELIABILITY ANALYST WITH THE CARE (COMPUTER-AIDED RELIABILITY
ESTIMATION) PACKAGE DO YOU WISH TO HAVE YOUR ANSWERS TO THE QUESTIONS PRINTED BACK.
ANSWER YES OR NO
YES
DO YOU WISH TO KNOW THE DEFINITIONS OF RELIABILITY PARAMETERS AND TERMS.
ANSWER YES OR NO
YES
THE DEFINITIONS OF THE VARIOUS RELIABILITY PARAMETERS AND TERMS ARE AS FOLLOWS.

      T = MISSION TIME.
      R = SYSTEM RELIABILITY.
      S = THE TOTAL NUMBER OF SPARES.
      N = THE NUMBER OF MULTIPLEXED UNITS.
      K = DORMANCY FACTOR = (LAMBDA/MU).
      C = COVERAGE FACTOR.
        = CONDITIONAL PROBABILITY OF SYSTEM RECOVERING GIVEN A FAILURE OCCURRENCE.
      Q = QUOTA, NUMBER OF IDENTICAL UNITS IN A SIMPLEX SYSTEM.
      W = NUMBER OF CASCADED UNITS.
      Z = NUMBER OF IDENTICAL SYSTEMS IN SERIES.
      P = PROBABILITY OF A UNIT FAILING TO A LOGIC ZERO.
     RV = RELIABILITY OF THE RESTORING ORGAN.
     MU = UNPOWERED FAILURE RATE OF A SIMPLEX SYSTEM = K/LAMBDA.
     LAMBDA = POWERED FAILURE RATE OF A SIMPLEX SYSTEM = K*MU.

     LAMT = NORMALIZED TIME = LAMBDA*MISSION TIME.
     ELAMT = EXP(-LAMT).
     REL = SYSTEM RELIABILITY.
     UNREL = SYSTEM UNRELIABILITY = (1 - REL).
     SIMREL = SIMPLEX RELIABILITY = ELAMT.
     SIMGAIN = GAIN IN RELIABILITY WITH REFERENCE TO A SIMPLEX SYSTEM.
             = REL/SIMREL.
     SIMRIF = RELIABILITY IMPROVEMENT FACTOR WITH REFERENCE TO A SIMPLEX SYSTEM.
            = (1 - SIMREL)/(1 - REL).

DO YOU NEED INSTRUCTIONS FOR RUNNING THE CARE PROGRAM
ANSWER YES OR NO
YES
SHORTCOMMENT — THE CARE PROGRAM COMPUTES, WITH RESPECT TO THE SELECTED EQUATIONS AND
PARAMETERS THE FOLLOWING RELIABILITY FUNCTIONS — THE RELIABILITY (REL), UNRELIABILITY
(UNREL), SIMPLEX RELIABILITY (SIMREL), SIMPLE GAIN (SIMGAIN), SIMPLE RELIABILITY IMPROVEMENT
FACTOR (SIMRIF), MEAN TIME TO FAILURE (MTF), RELIABILITY AT THE MTF, RELIABILITY DIFFERENCE
(DIFF), RELIABILITY GAIN (GAIN), RELIABILITY IMPROVEMENT FACTOR (RIF), SIMPLE MAXIMUM MISSION
TIME (SIMTMAX), MAXIMUM MISSION TIME (TMAX), SIMPLE TIME IMPROVEMENT FACTOR (SIMTIF), AND
THE RATIO OF TIME IMPROVEMENT FACTORS (RATIF).

2D AND SOME 3D PLOTS CAN BE OBTAINED FOR THE ABOVE COMPUTATIONS.
VARIOUS PLOTTING OPTIONS TO SPECIFY THE ABSCISSA, THE RANGE OF ABSCISSA AND ORDINATE VALUES
ARE AVAILABLE. ABILITY TO PLOT 3D INTERSECTIONS OF 3D PROJECTIONS WITH 2D PLANES IS ALSO AVAIL-
ABLE. THE CARE PROGRAM ALSO EVALUATES COMPLEX RELIABILITY FUNCTIONS FORMED BY TAKING
PRODUCTS OF THE BASIC RELIABILITY EQUATIONS.

CARE HAS A MAXIMUM OF 10 DIFFERENT RELIABILITY EQUATIONS. THESE ARE TABULATED BELOW.
 1. R(N,S) = F(T, LAMBDA, MU, S, N, K, RV, Z, W)
    THIS IS THE GENERAL RELIABILITY EQUATION OF AN HYBRID-REDUNDANT SYSTEM.
 2. R(Q,S) = F(T,LAMBDA,MU,S,K,Q,C,Z,W)
    THIS IS THE GENERAL RELIABILITY EQUATION OF A STANDBY-REPLACEMENT SYSTEM.
 3. VOID
 4. VOID
 5. R(3,0) = F(T,LAMBDA,RV,Z,W,P)
    THIS IS THE EQUATION FOR A TMR SYSTEM WHERE THE PROBABILITY OF A UNIT FAILING TO LOGICAL
    ONE OR ZERO IS PARAMETERIZED.
 6. R(1,0) = (EXP(-LAMBDA*T))**(Z/W)
    THIS IS A GENERAL EQUATION FOR A SIMPLEX SYSTEM.
 7. DUMMY
    THIS IS A DUMMY EQUATION WHICH IS ALL SET UP TO RECEIVE A NEW EQUATION.
 8. BLANK
 9. BLANK
10. BLANK

INSTRUCTIONS WILL BE GIVEN FOR ENTERING INPUT DATA AT THE TIME THE INPUT DATA IS NEEDED BY
THE PROGRAM.

DO YOU WISH TO FORM A PRODUCT OF RELIABILITIES
ANSWER YES OR NO
NO
TYPE IN COLUMN 1 THE NUMBER OF THE RELIABILITY EQUATION TO BE USED — 1 THRU 7

1
INPUT VARIABLES FOR EQUATION  1
T, LAMT, OR ELAMT MUST BE SPECIFIED AND ITS VALUE IS THE MAXIMUM VALUE FOR THAT VARIABLE.
MIN IS THE MINIMUM AND STEP IS THE INCREMENT FOR T, LAMT, OR ELAMT.
  SOME VARIABLES THAT ARE NEEDED BY THE EQUATIONS ARE SET EQUAL TO A DEFAULT VALUE IF THEY
ARE NOT INPUTTED. THESE VARIABLES AND THEIR DEFAULT VALUES ARE: S=1, N=1, Z=1, W=1, Q=1.0D0,
C=.999. . .DO, P=1.0D0, MIN=0.0D0, STEP=1.0D0, AND ELAMT=1.0D0.
  IF B IS INPUTTED, THEN THISVALUE IS USED AS THE FIRST GUESS FOR THE UPPER LIMIT OF INTEGRATION
IN THE CALCULATION OF MTF.
  IF OPTION=1, THEN DIFF, RIF, AND GAIN ARE CALCULATED FOR ALL POSSIBLE COMBINATIONS OF THE
PARAMETER. IF OPTION=2, THEN DIFF, RIF, AND GAIN ARE CALCULATED FOR THE LAST TWO PARAMETER
VALUES. IF OPTION=0 OR IS NOT INPUTTED, THEN THE PROGRAM WILL ASK THE USER AS TO WHICH PARAMETER
VALUES DIFF, RIF, AND GAIN ARE TO BE CALCULATED.
NOTE: DIFF, RIF, AND GAIN ARE NOT COMPUTED IF THE USER IS CALCULATING THE PRODUCT OF RELIABILITIES
OR PLOTTING 3-D. THE VARIABLES FOR EQUATION 1 ARE INPUTTED USING VAR AS THE NAMELIST NAME.
A SAMPLE INPUT FOR EQUATION 5 FOLLOWS:
  $VAR
  T=12.0D0,
  LAMBDA=1.0D0,1.5D0,2.0D0,
  RV=1.0D0,
  Z=1,
  W=1,6,
  OPTION=2
  B=10.0D0
  $END
NOTE: NAMELIST INPUT IGNORES COLUMN 1
THE INPUT VARIABLES ARE TYPED AS FOLLOWS
  DOUBLE PRECISION: T, LAMT, ELAMT, MUT, LAMBDA, MU, K, RV, Q, C, P, MIN, STEP, AND B
  INTEGER: S, N, W, Z, AND OPTION
INPUT VARIABLES NOW
DO YOU WISH TO MAKE ALTERATIONS TO THE $VAR LIST
ANSWER YES OR NO
NO
DO YOU WISH TO HAVE 2-D RELIABILITY PLOTS—ANSWER YES OR NO
YES
INPUT A 1 IN THE COLUMN SPECIFIED BELOW IF YOU WISH THE CORRESPONDING PLOT OPTION. OTHERWISE
INPUT O.
NOTE: WHEN PERFORMING PRODUCT OF RELIABILITIES, NO OTHER PLOT OPTION BESIDES PRODUCT OF RELIABILITIES
MAY BE SPECIFIED.
COLUMN 1—PLOTS PRODUCT OF RELIABILITIES
COLUMN 2—PLOTS RELIABILITY
COLUMN 3—PLOTS DIFF, RIF, AND GAIN
COLUMN 4—PLOTS MTF AND RELIABILITY AT MTF
COLUMN 5—PLOTS UNRELIABILITY
01100
```

```
FOR ABSCISSA, INPUT 1 IN COLUMN 1 IF ABSCISSA IS T,
1 IN COLUMN 2 IF ABSCISSA IS LOG(T)—BASE 10,
1 IN COLUMN 3 IF ABSCISSA IS LAMT,
1 IN COLUMN 4 IF ABSCISSA IS LOG(LAMT)—BASE 10,
1 IN COLUMN 5 IF ABSCISSA IS EXP(-LAMBDA*T),
1 IN COLUMN 6 IF ABSCISSA IS LOG(EXP(-LAMT))—BASE 10.
**1***
IF YOU WISH TO PLOT A CERTAIN RANGE OF X-AXIS VALUES FOR THE 2-D PLOTS, ENTER LEFT-END POINT IN
COLUMNS 1–8 WITH FORMAT F8.0 AND RIGHT-END POINT IN COLUMNS 9–16 WITH FORMAT F8.0; OTHERWISE
INPUT NO
NO
IF YOU WISH TO PLOT A CERTAIN RANGE OF Y-AXIS VALUES FOR THE 2-D PLOTS, ENTER LEFT-END POINT
IN COLUMNS 1–8 WITH FORMAT F8.0 AND RIGHT-END POINT IN COLUMNS 9–16 WITH FORMAT F8.0; OTHERWISE
INPUT NO
NO
DO YOU WISH TO PLOT THE LOCUS OF RV SUCH THAT THE SYSTEM RELIABILITY EQUALS THE UNIT RELI-
ABILITY.
ANSWER YES OR NO
NO
DO YOU WISH TO HAVE 3-D RELIABILITY PLOTS—ANSWER YES OR NO
NO
DO YOU WISH TO CALCULATE MAXIMUM MISSION TIME AND SIMPLE TIME FOR GIVEN RELIABILITY—ANSWER
YES OR NO
YES
DO YOU WANT PLOTS FOR THESE CALCULATIONS—ANSWER YES OR NO
YES
DO YOU WISH TO CALCULATE MAXIMUM MISSION TIME FOR GIVEN RELIABILITY AND COMPARE IT AGAINST
OTHER PARAMETERS
ANSWER YES OR NO
YES
INPUT IN COLUMN 1 ONE OF THE FOLLOWING THREE OPTIONS:
1. MAXIMUM MISSION TIME IS COMPARED AGAINST ALL POSSIBLE COMBINATIONS OF THE PARAMETER,
2. MAXIMUM MISSION TIME IS COMPARED AGAINST THE LAST TWO PARAMETER VALUES,
3. THE PROGRAM ASKS THE USER AS TO WHICH PARAMETER VALUES MAXIMUM MISSION TIME IS TO BE
   COMPARED.
1
DO YOU WANT PLOTS FOR THESE CALCULATIONS—ANSWER YES OR NO
NOTE: WHEN EXERCISING OPTION 1, THE PROGRAM PLOTS ONLY THE FIRST 15 PARAMETER COMPARISONS
YES
INPUT THE FOLLOWING 4 VARIABLES EACH WITH FORMAT F8.0
COLUMNS  1–8 —REFERENCE RELIABILITY R2
COLUMNS  9–16—MINIMUM RELIABILITY R1
COLUMNS 17–24—MAXIMUM RELIABILITY R1
COLUMNS 25–32—RELIABILITY R1 STEP SIZE
   1.000      .000      1.000      .100
DO YOU WISH TO HAVE PRINTED TABLE OF RELIABILITY RESULTS
ANSWER YES OR NO
YES
DO YOU WISH TO HAVE PRINTED TABLE OF DIFF, RIF, AND GAIN RESULTS—ANSWER YES OR NO
YES
DO YOU WISH MTF AND RELIABILITY AT MTF RESULTS PRINTED
ANSWER YES OR NO
YES
DO YOU WANT PRINTED RESULTS OF THE MAXIMUM MISSION TIME CALCULATIONS—ANSWER YES OR NO
YES
TYPE IN THE VARIABLE THAT IS TO BE USED FOR THE FAMILY OR PARAMETERS—MUST BE SPECIFIED
K
```

```
               CALCULATIONS FOR EQUATION 1A      (NI MEANS NOT INPUTTED)
                                 PARAMETER IS K
-------------------------------------------------------------------------------
 LAMBDA          MU              S              N             K              Q
   NI        .0000000            1              1        .1000000+01        NI
   C            RV               Z              W             P             MUT
   NI        .1000000+01         1              1        .1000000+01        NI
-------------------------------------------------------------------------------
  LAMT          REL          UNREL         SIMREL        SIMGAIN        SIMRIF
   .000      1.0000000      .0000000      1.0000000     .1000000+01    .1000000+36
   .100       .9967989      .0032011       .9048374     .1101633+01    .2972798+02
   .200       .9794141      .0205959       .8187307     .1196259+01    .8805495+01
   .300       .9438952      .0561048       .7408182     .1274125+01    .4619598+01
   .400       .8921096      .1078904       .6703200     .1330871+01    .3055694+01
   .500       .8282412      .1717588       .6065307     .1365539+01    .2290825+01
   .600       .7569280      .2430720       .5488116     .1379213+01    .1856192+01
   .700       .6823605      .3176395       .4965853     .1374105+01    .1584862+01
   .800       .6079221      .3920779       .4493290     .1352955+01    .1404494+01
   .900       .5361204      .4638796       .4065697     .1318643+01    .1279277+01
  1.000       .4686621      .5313379       .3678794     .1273956+01    .1189677+01
  1.100       .4065856      .5934144       .3328711     .1221451+01    .1124221+01
  1.200       .3504072      .6495928       .3011942     .1163393+01    .1075760+01
  1.300       .3002559      .6997441       .2725318     .1101728+01    .1039620+01
  1.400       .2559894      .7440107       .2465970     .1038088+01    .1012624+01
  1.500       .2172867      .7827133       .2231302     .9738114-00    .9925343-00
  1.600       .1837199      .8162801       .2018965     .9099707-00    .9777324-00
  1.700       .1548070      .8451930       .1826835     .8474052-00    .9670175-00
  1.800       .1300494      .8699506       .1652989     .7867533-00    .9594811-00
  1.900       .1089583      .8910417       .1495686     .7284834-00    .9544237-00
  2.000       .0910702      .9089298       .1353353     .6729228-00    .9512998-00
  2.100       .0759576      .9240424       .1224564     .6202829-00    .9496789-00
  2.200       .0632333      .9367667       .1108032     .5706813-00    .9492191-00
  2.300       .0525518      .9474482       .1002588     .5421617-00    .9496468-00
  2.400       .0436090      .9563910       .0907180     .4807095-00    .9507430-00
  2.500       .0361392      .9638608       .0820850     .4402657-00    .9523315-00
  2.600       .0299128      .9700872       .0742736     .4027382-00    .9542714-00
  2.700       .0247324      .9752676       .0672055     .3680110-00    .9564498-00
  2.800       .0204293      .9795707       .0608101     .3359521-00    .9587770-00
  2.900       .0168601      .9831399       .0550232     .3064186-00    .9611824-00
  3.000       .0139037      .9860963       .0497871     .2792626-00    .9636107-00
-------------------------------------------------------------------------------
MEAN TIME TO FAILURE - MTF = .10833333+01.
UPPER LIMIT FOR INTEGRATION - B = .15000000+02
RELIABILITY AT MTF = .41653059-00

     MAXIMUM MISSION
            TIME            REFERENCE R2 = 1.0000
    R1       SIMLAMTMAX       LAMTMAX         SIMTIF
   .00000     INFINITY        INFINITY      .1000000+01
   .10000    .2302585+01    .1948467+01    .8462084-00
   .20000    .1609438+01    .1549781+01    .9629332-00
   .30000    .1203973+01    .1300594+01    .1080252+01
   .40000    .9162907-00    .1111202+01    .1212718+01
   .50000    .6931472-00    .9526588-00    .1374396+01
   .60000    .5108256-00    .8108549-00    .1587342+01
   .70000    .3566749-00    .6764670-00    .1896592+01
   .80000    .2231436-00    .5404841-00    .2422136+01
   .90000    .1053605+00    .3862209-00    .3665708+01
  1.00000    .0000000       .0000000       .1000000+01
TMAX AND SIMTIF PLOT COMPLETED
TMAX AND SIMIF PLOT COMPLETED
MAXIMUM MISSION TIME FOR K     = .1000000+001
AND K     = .1000000+006     FOLLOWS FOR EQUATION    1B
               REFERENCE R2 = 1.00000

    R1             TMAX1              TMAX2       RATIF

    .00000     INFINITY          INFINITY          .1000000+01
    .10000     .1948467+01      .2083571+01       .1069339+01
    .20000     .1549781+01      .1666156+01       .1075091+01
    .30000     .1300594+01      .1403234+01       .1078918+01
    .40000     .1111202+01      .1202074+01       .1081777+01
    .50000     .9526588-00      .1033006+01       .1084340+01
    .60000     .8108549-00      .8812095-00       .1086766+01
    .70000     .6764670-00      .7366257-00       .1088931+01
    .80000     .5404841-00      .5897715-00       .1091191+01
    .90000     .3862209-00      .4224357-00       .1093767+01
   1.00000     .0000000         .0000000          .1000000+01

  1 MAXIMUM MISSION TIME PLOTS FOR VARYING
  PARAMETER VALUES COMPLETED
  DIFF, RIF, AND GAIN FOR K     = .1000000+001
  AND K     = .1000000+006     FOLLOWS FOR EQUATION   1B

           LAMT              DIFF            RIF              GAIN

           .00000          .000000        INFINITY          .100000+01
           .10000          .741191-03     .130131+01        .100074+01
           .20000          .439928-02     .127178+01        .100449+01
           .30000          .110269-01     .124462+01        .101168+01
           .40000          .194312-01     .121966+01        .102178+01
           .50000          .282420-01     .119679+01        .103410+01
           .60000          .363528-01     .117486+01        .104803+01
           .70000          .430437-01     .115674+01        .106308+01
           .80000          .479568-01     .113936+01        .107889+01
           .90000          .510157-01     .112357+01        .109516+01
          1.00000          .523365-01     .110926+01        .111167+01
          1.10000          .521486-01     .109635+01        .112826+01
          1.20000          .507338-01     .108472+01        .114470+01
          1.30000          .483841-01     .107428+01        .116114+01
          1.40000          .453738-01     .106495+01        .117725+01
          1.50000          .419440-01     .195662+01        .119304+01
          1.60000          .382964-01     .104923+01        .120845+01
          1.70000          .345917-01     .104267+01        .122345+01
          1.80000          .309523-01     .103689+01        .123800+01
          1.90000          .274670-01     .103181+01        .125209+01
          2.00000          .241955-01     .102735+01        .126568+01
          2.10000          .211747-01     .102345+01        .127877+01
          2.20000          .184230-01     .102006+01        .129135+01
          2.30000          .159450-01     .101712+01        .130341+01
          2.40000          .137352-01     .101457+01        .131496+01
          2.50000          .117814-01     .101237+01        .132600+01
          2.60000          .100665-01     .101049+01        .133653+01
          2.70000          .857121-02     .100887+01        .134656-01
          2.80000          .727480-01     .100748+01        .135610+01
          2.90000          .615657-02     .100630+01        .136516+01
          3.00000          .519645-02     .100530+01        .137375+01

  1 PLOTS COMPLETED
  3 PLOTS COMPLETED
  DO YOU WISH TO SPECIFY ANOTHER PARAMETER
  ANSWER YES OR NO
  NO
  —FIN
```

CALCULATIONS FOR EQUATION 1B (NI MEANS NOT INPUTTED)
PARAMETER IS K

LAMBDA	MU	S	N	K	Q
NI	NI	1	1	INF	NI
C	RV	Z	W	P	MUT
NI	.1000000+01	1	1	.1000000+01	NI

LAMT	REL	UNREL	SIMREL	SIMGAIN	SIMRIF
.000	1.0000000	.0000000	1.0000000	.1000000+01	.1000000+36
.100	.9975401	.0024599	.9048374	.1102452+01	.3868510+02
.200	.9838134	.0161866	.8187370	.1201632+01	.1119870+02
.300	.9549221	.0450779	.7408182	.1289010+01	.5749636+01
.400	.9115409	.0884591	.6703200	.1359859+01	.3726918+01
.500	.8564832	.1435168	.6065307	.1412102+01	.2741626+01
.600	.7932808	.2067192	.5488116	.1445452+01	.2182615+01
.700	.7254042	.2745958	.4965853	.1460785+01	.1833294+01
.800	.6558789	.3441211	.4493290	.1459685+01	.1600224+01
.900	.5871361	.4128639	.4065697	.1444122+01	.1437351+01
1.000	.5209986	.4790014	.3678794	.1416221+01	.1319663+01
1.100	.4587342	.5412658	.3328711	.1378114+01	.1232535+01
1.200	.4011410	.5988590	.3011942	.1331835+01	.1166895+01
1.300	.3486400	.6513600	.2725318	.1279264+01	.1116845+01
1.400	.3013631	.6986369	.2465970	.1222088+01	.1078390+01
1.500	.2592307	.7407693	.2231302	.1161791+01	.1048734+01
1.600	.2220163	.7779837	.2018965	.1099654+01	.1025861+01
1.700	.1893986	.8106014	.1826835	.1036758+01	.1008284+01
1.800	.1610018	.8389982	.1652989	.9740040−00	.9948783−00
1.900	.1364252	.8635748	.1495686	.9121246−00	.9847802−00
2.000	.1152657	.8847343	.1353353	.8517048−00	.9773157−00
2.100	.0971323	.9028677	.1224564	.7931990−00	.9719515−00
2.200	.0816563	.9183438	.1108032	.7369488−00	.9682614−00
2.300	.0684968	.9315032	.1002588	.6831996−00	.9659024−00
2.400	.0573442	.9426558	.0907180	.6321152−00	.9645960−00
2.500	.0479206	.9520794	.0820850	.5837922−00	.9641160−00
2.600	.0399793	.9600207	.0742736	.5382713−00	.9642776−00
2.700	.0333036	.9666964	.0672055	.4955484−00	.9649301−00
2.800	.0277041	.9722959	.0608101	.4555835−00	.9659507−00
2.900	.0230167	.9769833	.0550232	.4183091−00	.9672395−00
3.000	.0191001	.9808999	.0497871	.3836361−00	.9687155−00

MEAN TIME TO FAILURE − MTF = .11666667+01
UPPER LIMIT FOR INTEGRATION − B = .15000000+02
RELIABILITY AT MTF = .41978696−00

MAXIMUM MISSION
TIME REFERENCE R2 = 1.00000

R1	SIMLAMTMAX	LAMTMAX	SIMTIF
.00000	INFINITY	INFINITY	.1000000+01
.10000	.2302585+01	.2083571+01	.9048836−00
.20000	.1609438+01	.1666156+01	.1035241+01
.30000	.1203973+01	.1403234+01	.1165503+01
.40000	.9162907−00	.1202074+01	.1311891+01
.50000	.6931472−00	.1033006+01	.1490313+01
.60000	.5108256−00	.8812095−00	.1725069+01
.70000	.3566749−00	.7366257−00	.2065258+01
.80000	.2231436−00	.5897715−00	.2643014+01
.90000	.1053605+00	.4224357−00	.4009430+01
1.00000	.0000000	.0000000	.1000000+01

An adaptive error correction scheme for computer memory system

by A. M. PATEL and M. Y. HSIAO

IBM Corporation
Poughkeepsie, New York

INTRODUCTION

Many of the modern computer memories contain single-error correction capability in order to enhance reliability.[1] In a large scale memory, an even more powerful error correction code may be desirable. In particular, a double-error correction capability can reduce the maintenance cost significantly, while keeping the unscheduled system interruptions within tolerable limits. Since most faults are effectively masked and logged out, the permanent failures can be replaced at the time of scheduled maintenance, thus leaving the user unaffected. The cost and complexity of the known double error correcting code, however, seems to outweigh the advantages. The long decoding time and large amount of redundancy in double error correction cannot be justified in *every* fetch instruction for the sake of correcting an *occasional* double error.

This paper describes a memory error correction scheme which can be used in an adaptive manner. The code used in this scheme is derived from a full length BCH double error correcting code[2] by deleting certain columns of the parity check matrix. This code corrects single errors as well as double errors on different memory word boundaries while the number of check bits required is much less and the normal memory cycle remains unaffected except in the presence of a double error.

GENERAL SYSTEM FEATURES

In this section, the adaptive error correction scheme is illustrated through an example. Let us assume that the word length of a basic memory unit is 64 data bits plus 8 check bits for single error correction and double error detection. If the double error correction (DEC) feature using BCH code is desired, then additional 7 check bits are required. Since a memory system may have several, say m, basic memory units rather than one, it requires $7\times m$ extra check bits for DEC in the total system. This results into high cost of implementation and also increases the memory cycle time even if only a single error has occurred. The adaptive ECC scheme guarantees the SEC-DED capability on each basic memory unit and only uses 7 extra check bits or 8 bits if an overall triple error detection (TED) capability is required for DEC on the entire system. Single errors are corrected without referring to the additional check bits, hence nominal memory cycle time is not affected. Only in the case of double errors, the memory cycle time is increased. The parity check matrix has the following form:

$$H=\begin{bmatrix} [H_{64} & I_8] & \cdots\cdots & \phi & \phi & \phi \\ & \ddots & & & \\ \phi & \phi & \cdots\cdots & [H_{64} & I_8] & \phi \\ [A & \phi] & \cdots\cdots & [A & \phi] & I_8 \end{bmatrix} \tag{1}$$

The construction of the submatrices H_{64} and A is done by an APL program[3] given in the appendix with theory stated in Section III. The submatrix ϕ is a null matrix of all zeros and I_8 is a 8×8 identity matrix.

In the memory system, each word of 64 information bits independently carries 8 check bits which provide the SEC-DED capability on every word separately. Thus, any single error in any word can be corrected separately without any reference to other words. However, in the presence of two errors in one word (DED indicator), one will compute the error pattern from the correlation of the SEC 7-bit syndrome of the erroneous word with the DEC 7-bit syndrome computed over all m contributing words. Any single error in other words is either eliminated separately before computing the DEC syndrome or is detected by the TED check bit. If two or more words in a group indicate double errors, DEC syndrome will be a composite of two syndromes and, hence, the errors in such case cannot be corrected. However, the probability of a double-error being very low, two errors in each of the two words in a group, is highly improbable. Hence, in most cases the double error in a word will be correctible.

CONSTRAINTS FOR THE MODIFIED DEC BCH CODES

The double error correcting code for such adaptive correction scheme must have the following two features:

(1) The 14 check bits are divided into two groups of 7 check bits each. Each group must be independent of the other.
(2) The parity rules for the first group must satisfy the constraints of the SEC-DED code and the parity rules of the two groups together must provide the DEC-TED capability.

We examine the double-error correcting BCH code for these two SEC-separability properties:

There are at least three methods of generating the parity check matrix of a double error correcting code. The parity check matrix denoted by H_1 has $x^i \bmod m_1(x)m_3(x)$ as its ith column (0 origin) where $m_1(x)$ and $m_3(x)$ are minimum functions of the field elements α and α^3 of GF (2^7). The parity check matrix H_2 generated by the second method has the concatenated vector $x^i \bmod m_1(x)$, $x^i \bmod m_3(x)$ as its ith column. The parity check matrix H_3 generated by the third method has the concatenated vector $x^i \bmod m_1(x)$, $x^{3i} \bmod m_1(x)$ as its ith column. The codes generated by these three matrices are not only equivalent but also isomorphic. These three matrices possess different desirable properties. In particular, the matrix H_1 possess the property (1) for the adaptive correction scheme—presently under consideration. The first 14 columns of H_1 represent an identity matrix which corresponds to 14 independently-acting check digits. However, any 7 check bits as a group do not provide SEC capability which is the required property (2). The matrix H_2 on the other hand can be divided into two parts where the first group of seven check bits, corresponding in the part column $x^i \bmod m_1(x)$, does provide SEC capability, however, the two groups of check bits do not act independently and hence are not separable. The matrix H_3 behaves in the same manner as H_2 except that the syndrome in H_3 is easily decodable.[4]

As it seems from the above discussions, the full length DEC code does not possess the SEC—separability properties (1) and (2). However, it is easy to see that one could drop a number of columns from matrix H_1 in order to obtain the SEC capability with the first seven check bits. We examine the first seven digits of each column $i(i>13)$ and drop the column if this seven digit vector has already appeared in a previously taken column. This guarantees that these columns along with the first 7 columns for check bits form a single error correcting code. This exercise was carried out using an APL computer program which generated a (104, 90) and (172, 154) DEC codes which has separable SEC and can be shortened to handle data bit lengths 64 and 128. The codes are given in the Appendix. The DED capability is obtained by adding a check bit on the SEC code which makes a SEC-DED odd-weight-column-code.[5] The number in front of each column of H-matrix in the Appendix represents the cyclic position number in the full length code. These position numbers are used in the algebraic decoding algorithm[4] in error correction process.

SYSTEM IMPLEMENTATION

Let us use a simple example for illustration. Figure 1 shows a memory system which contains two basic memory units. Each unit has a (72, 64) SEC-DED code.

The following is the parity check matrix for this simple system.

$$H=\begin{bmatrix} [H_{64} & I_8] & \phi & \phi & \phi \\ \phi & \phi & [H_{64} & I_8] & \phi \\ [A & \phi] & [A & \phi] & I_8 \end{bmatrix} \qquad (2)$$

Where H_{64} is the first group of 7 columns of the matrix in the Appendix and an additional column is added to make it odd weight. The A-matrix is the second group of 7 columns of the matrix in the Appendix. Another column is added to these 15 columns to make the overall parity matrix odd weight. This means that the overall code has double error correction and triple error detection capability. The encoding follows directly from the H-matrix of Equation (2). The decoding is classified as follows:

1. Any single error in each memory unit can be corrected separately and simultaneously.
2. If a double error is detected in one of the memory units and no error indication in the other memory

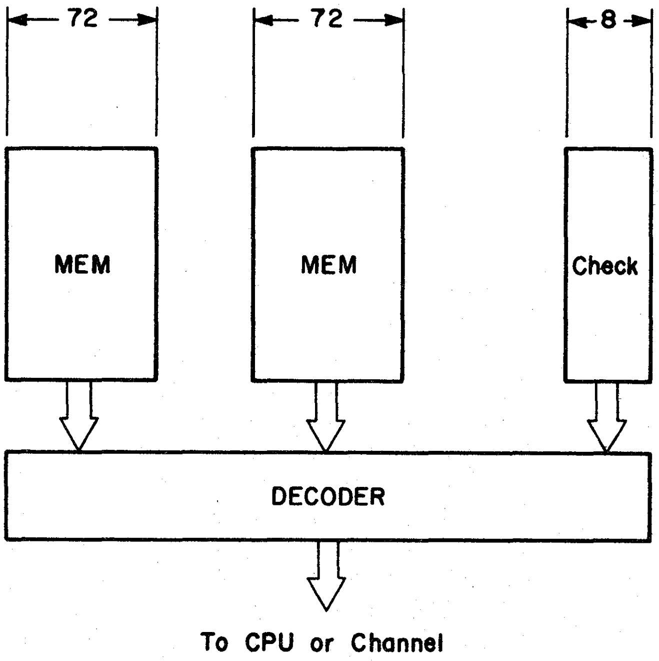


Figure 1

unit for the corresponding word, then this double error can be corrected by the additional 8 check bits.

The decoding of the double errors as stated in class 2 needs the data bits portion of both memory units. The data bit portion for the error free memory is required to cancel its effects in the last 8 syndrome bits. Therefore, the double error correction can be done as that given in Reference 4.

SUMMARY

An adaptive ECC scheme with SEC-DED feature can be expanded to DEC feature in a memory system containing several memory units environment. The normal memory cycle time remains unaffected, except in the presence of a double error when extra decoding time is required for the double error correction procedure. Other major advantage is cost savings in terms of number of check bits required. If the memory system contains m basic memory units then $8(m-1)$ check bits can be saved by using this scheme. The number m is chosen such that the probability of double-errors in two words out of a group of m words is very small. Such adaptive error correction scheme more closely matches the requirements of modern computer memory systems and can be used very effectively for masking faults and reducing cost of maintenance.

REFERENCES

1 J F KEELEY
System/370—Reliability from a system viewpoint
1971 IEEE International Computer Society Conference September 22-24 1971 Boston Massachusetts

2 W W PETERSON
Error correcting codes
MIT Press 1961

3 A D FALKOFF K E IVERSON
APL/360 user's manual
IBM Watson Research Center Yorktown Heights New York 1968

4 A M PATEL S J HONG
Syndrome trapping technique for fast double-error correction
Presented at the 1972 IEEE International Symposium on Information Theory IEEE Catalog No 72 CHO 578-S IT 1972

5 M Y HSIAO
A class of optimal minimum odd-weight-column SEC-DED codes
IBM J of Res & Dev Vol 14 No 4 pp 395-402 July 1970

APPENDIX A—CODE GENERATION PROGRAM

```
                    APL 360


          ∇SECDEC[⎕]∇


        ∇ SECDEC G
[1]       M←¯1+ρG
[2]       N←2*(M÷2)
[3]       V←Mρ0
[4]       Z←Nρ0
[5]       Q←(Mρ0),1
[6]       I←0
[7]       V←MρQ←(¯1⌽Q)≠(G×Q[M-1])
[8]       →9+(I>M)×((X←(2⊥((M÷2)↑V)))∊Z)
[9]       '0123456789*'[(10 10 10 ⊤I),10,V]
[10]      Z[I]←X
[11]      I←I+1
[12]      →7+(7×(I=N))
[13]      'DONE'
[14]      →15
        ∇
```

APPENDIX B—PARITY CHECK MATRIX FOR (104, 90) SEC-SEPARABLE DEC CODE

```
SECDEC 1 0 0 0 0 1 1 0 1 1 1 0 1 1 1
```

```
000*10000000000000
001*01000000000000
002*00100000000000
003*00010000000000
004*00001000000000
005*00000100000000
006*00000010000000
007*00000001000000
008*00000000100000
009*00000000010000
010*00000000001000
011*00000000000100
012*00000000000010
013*00000000000001
014*10000110111011
015*11000101100110
016*01100010110011
017*10110111100010
018*01011011110001
019*10101011000011
020*11010011011010
021*01101001101101
022*10110010001101
023*11011111111101
024*11101001000101
025*11110010011001
026*11111111110111
027*11111001000000
028*01111100100000
029*00111110010000
030*00011111001000
031*00001111100100
032*00000111110010
037*00110001010110
038*00011000101011
039*10001010101110
040*01000101010111
041*10100100010000
042*01010010001000
043*00101001000100
044*00010100100010
045*00001010010001
046*10000011110011
047*11000111000010
050*11011101011110
051*01101110101111
052*10110001101100
053*01011000110110
054*00101100011011
055*10010000110110
056*01001000011011
057*10100010110110
058*01010001011011
059*10101110010110
060*01010111001011
061*10101101011110
065*00101011011011
066*10010011010110
074*10001100000010
075*01000110000001
077*11010100000110
078*01101010000011
083*11101111000111
084*11110001011000
085*01111000101100
086*00111100010110
088*10001001111110
094*11001111110100
095*01100111111010
096*00110011111101
097*10011111000101
098*11001001011001
099*11100010010111
100*11110111110000
101*01111011111000
108*10010110110001
109*11001101100011
110*11100000001010
111*01110000000101
112*10111110111001
113*11011001100111
114*11101010001000
115*01110101000100
116*00111010100010
117*00011101010001
119*11000010110010
120*01100001011001
124*00110111011100
125*00011011101110
126*00001101110111
```

APPENDIX C—(172, 154) SEC-SEPARABLE DEC CODE

```
SEC DEC
SECDEC 1 0 1 1 0 1 1 1 1 0 1 1 0 0 0 1 1

000*1000000000000000
001*0100000000000000
002*0010000000000000
003*0001000000000000
004*0000100000000000
005*0000010000000000
006*0000001000000000
007*0000000100000000
008*0000000010000000
009*0000000001000000
010*0000000000100000
011*0000000000010000
012*0000000000001000
013*0000000000000100
014*0000000000000010
015*0000000000000001
016*1011011110110001
017*1110110001101001
018*1100000110000101
019*1101011101110011
020*1101110000001000
021*0110111000000100
022*0011011100000010
023*0001101110000001
024*1011101001110001
025*1110101010001001
026*1100001011110101
027*1101011011001011
031*1010110000101011
032*1110000110100100
033*0111000011010010
034*0011100001101001
035*1010101110000101
036*1110001001110011
037*1100011010001000
038*0110001101000100
039*0011000110100010
040*0001100011010001
041*1011101111011001
045*0110101101111111
046*1000001000001110
047*0100000100000111
048*1001011100110010
049*0100101110011001
050*1001001001111101
051*1111111010001111
052*1100100011110110
053*0110010001111011
054*1000010110001100
055*0100001011000110
056*0010000101100011
057*1010011100000000
058*0101001110000000
059*0010100111000000
060*0001010011100000
061*0000101001110000
062*0000010100111000

067*0101101111110001
068*1001101001001001
069*1111101010010101
070*1100101011111011
071*1101001011001100
072*0110100101100110
073*0011010010110011
074*1010110111101000
075*0101011011110100
076*0010101101111010
077*0001010110111101
078*1011110101101111
079*1110100100000110
080*0111010010000011
081*1000110111110000
082*0100011011111000
083*0010001101111100
084*0001000110111110
086*1011001111011110
087*0101100111101111
088*1001101101000110
089*0100110110100011
090*1001000101100000
091*0100100010110000
092*0010010001011000
093*0001001000101100
094*0000100100010110
096*1011010111110100
097*0101101011111010
098*0010110101111101
099*1010000100001111
100*1110011100110110
101*0111001110011011
102*1000111001111100
103*0100011100111110
105*1010011001111110
107*1001111000101110
108*0100111100010111
109*1001000000111010
111*1001001110111111
113*0111111100110111
114*1000100000101010
115*0100010000010101
116*1001010110111011
117*1111110101101100
118*0111111010110110
119*0011111101011011
120*1010100000011100
121*0101010000001110
122*0010101000000111
123*1010001010110010
124*0101000101011001
125*1001111100011101
126*1111100000111111
127*1100101110101110
128*0110010111010111
131*1001011011100111
132*1111110011000010

135*1111001111110001
136*1100111001001001
137*1101000010010101
138*1101111111111011
139*1101100001001100
140*0110110000100110
141*0011011000010011
147*0101111010111101
148*1001100011101111
149*1111101111000110
150*0111110111100011
151*1000100101000000
153*0010001001010000
158*1011011010100011
160*0111011001110000
161*0011101100111000
162*0001110110011100
163*0000111011001110
164*0000011101100111
165*1011010000000010
171*1101111011011000
172*0110111101101100
176*0101110100101110
177*0010111010010111
178*1010000011111010
179*0101000001111101
182*0111110000111011
189*1010001101100001
190*1110011000000001
191*1100010010110001
192*1101010111101001
193*1101110101000101
194*1101100100010011
195*1101101100111000
196*0110110110011100
201*1001100100110001
209*0000110111101110
210*0000011011110111
216*0011111010111100
217*0001111101011110
218*0000111110101111
219*1011000001100110
220*0101100000110011
223*0010011011101010
224*0001001101110101
225*1011111000001011
226*1110100010110100
228*0011101000101101
229*1010101010100111
231*0111000101110001
232*1000111100001001
233*1111000000110101
234*1100111110101011
235*0110100000110010
240*1100011100000110
242*1000011001110000
243*0100001100111000
```

Dynamic confirmation of system integrity*

by BARRY R. BORGERSON

University of California
Berkeley, California

INTRODUCTION

It is always desirable to know the current state of any system. However, with most computing systems, a large class of failures can remain undetected by the system long enough to cause an integrity violation. What is needed is a technique, or set of techniques, for detecting when a system is not functioning correctly. That is, we need some way of observing the integrity of a system.

A slight diversion is necessary here. Most nouns which are used to describe the attributes of computer systems, such as reliability, availability, security, and privacy, have a corresponding adjective which can be used to identify a system that has the associated attribute. Unfortunately, the word "integrity" has no associated adjective. Therefore, in order to enhance the following discourse, the word "integral" will be used as the adjective which describes the integrity of a system. Thus, a computer system will be integral if it is working exactly as specified.

Now, if we could verify all of the system software, then we could monitor the integrity of a system in real time by providing a 100 percent concurrent fault detection capability. Thus, the integrity of the entire system would be confirmed concurrently, where "concurrent confirmation" of the integrity of any unit of logic means that the integrity of this unit is being monitored concurrently with each use.

A practical alternative to providing concurrent confirmation of system integrity is to provide what will be called "dynamic confirmation of system integrity." With this concept, the parts of a system that *must* be continuously integral are identified, and the integrity of the rest of the system can then be confirmed by means less stringent than concurrent fault detection. For example, it might be expedient to allow certain failures to exist for some time before being detected. This might be desirable, for instance, when certain failure modes are hard to detect concurrently, but where their effects are controllable.

QUALITATIVE JUSTIFICATION

In most contemporary systems, a multiplicity of processes are active at any given time. Two distinct types of integrity violations can occur with respect to the independent processes. One type of integrity violation is for one process to interfere with another process. That is, one process gains unauthorized access to another's information or makes an illegitimate change of another process' state. This type of transgression will be called an "interprocess integrity violation." The other basic type of malfunction which can be caused by an integrity violation occurs when the state of a single process is erroneously changed without any interference from another process. Failures which lead to only intraprocess contaminations will be called "intraprocess integrity violations."

For many real-time applications, no malfunctions of any type can be tolerated. Hence, it is not particularly useful to make the distinction between interprocess and intraprocess integrity violations since concurrent integrity-confirmation techniques must be utilized throughout the system. For most user-oriented systems, however, there is a substantial difference in the two types of violations. Intraprocess integrity violations always manifest themselves as contaminations of a process' environment. Interprocess integrity violations, on the other hand, may manifest themselves as security infractions or contaminations of other processes' environments.

* This research was supported by the Advanced Research Projects Agency under contract No. DAHC15 70 C 0274. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

We now see that there can be some freedom in defining what is to constitute a continuously-integral, user-oriented system. For example, the time-sharing system described below is defined to be continuously integral if it is providing interprocess-interference protection on a continuous basis. Thus other properties of the system, such as intraprocess contamination protection, need not be confirmed on a continuous basis.

Although the concept of dynamic confirmation of system integrity has a potential for being useful in a wide variety of situations, the area of its most obvious applicability seems to be for fault-tolerant systems. More specifically, it is most useful in those systems which are designed using a solitary-fault assumption. Where "solitary fault" means that at most one fault is present in the active system at any time. The notion of "dynamic" becomes more clear in this context. Here, "dynamic" means in such a manner, and at such times, so that the probability of encountering simultaneous faults is below a predetermined limit. This limit is dictated not only by the allowable probability of a catastrophic failure, but also by the fact that other factors eventually become more prominent in determining the probability of system failure. Thus, there often becomes a point beyond which there is very little to be gained by increasing the ability to confirm integrity. The rest of this paper will concern itself with dynamic confirmation in the context of making this concept viable with respect to the solitary-fault assumption.

DYNAMIC CONFIRMATION TECHNIQUES

In this section, and the following section, a particular class of systems will be assumed. The class of systems considered will be those which tolerate faults by restructuring to run without the faulty units. Both the stand-by sparing and the fail-softly types of systems are in this category. These systems have certain characteristics in common; namely, they both must detect, locate, and isolate a fault, and reconfigure to run without the faulty unit, before a second fault can be reliably handled.

Obviously, if simultaneous faults are to be avoided, the integrity of all parts of the system must be verified. This is reasonably straightforward in many areas. For instance, the integrity of data in memory can be rather easily confirmed by the method of storing and checking parity. Of course, checks must also be provided to make sure that the correct word of memory is referenced, but this can be done fairly easily too.[1] It is generally true that parity, check sums, and other straightforward concurrent fault-detection techniques can be used to confirm the integrity of most of the logic external to processors. However, there still remains the problems of verifying the integrity of the checkers themselves, of the processors, and of logic that is infrequently used such as that associated with isolation and reconfiguration.

All too often, there is no provision made in a system to check the fault detection logic. Actually, there are two rather straightforward methods of accomplishing this. One method uses checkers that have their own failure space. That is, they have more than two output states; and when they fail, a state is entered which indicates that the checker is malfunctioning. This requires building checkers with specifically defined failure modes. It also requires the ability to recognize and handle this limbo state. An example of this type of checker appears in Reference 2.

Another method for verifying the integrity of the fault-detection logic is to inject faults; that is, cause a fault to be created so that the checker must recognize it. In many cases this method turns out to be both cheaper and simpler than the previously mentioned scheme. With this method, it is not necessary to provide a failure space for the checkers themselves.However, it is necessary to make provisions for injecting faults when that is not already possible in the normal design. With this provision, confirming the integrity of the checking circuits becomes a periodic software task. Failures are injected, and fault detection inputs are expected. The system software simply ignores the fault report or initiates corrective action if no report is generated.

Associated with systems of the type under discussion, there is logic that normally is called into use only when a fault has been detected. This includes the logic dedicated to such tasks as diagnosis, isolation, and reconfiguration. This normally idle class of hardware units will collectively be called "reaction logic." In order to avoid simultaneous faults in a system, this reaction logic must not be allowed to fail without the failure being rapidly detected. Several possibilities exist here. This logic can be made very reliable by using some massive redundancy technique such as triple-modular-redundancy.[3] Another possibility is to design these units such that they normally fail into a failure space which is detected and reported. However, this will not be as simple here as it might be for self-checking fault detectors because the failure modes will, in general, be harder to control. A third method would be to simulate the appropriate action and observe the reaction. This also is not as simple here as it was above. For example, it may not be desirable to reconfigure a system on a frequent periodic basis. However, one way out of this is to simulate the action, initiate the reaction, and confirm the integrity of this logic without actually causing the reconfiguration. This will probably require that the output logic either be made "reliable" or be encoded so as to fail into a harmless and detectable failure space.

The final area that requires integrity confirmation is the processors. The technique to be employed here is very dependent on the application of the system. For many real-time applications, nothing short of concurrent fault detection will apparently suffice. However, there are many areas where less drastic methods may be adequate. Fabry[4] has presented a method for verifying critical operating-system decisions, in a time-sharing environment, through a series of independent double checks using a combination of a second processor and dedicated hardware. This method can be extended to verifying certain decisions made by a real-time control processor. If most of the tasks that a real-time processor performs concern data reduction, it is possible that software-implemented consistency checks will suffice for monitoring the integrity of the results. When critical control decisions are to be made, a second processor can be brought into the picture for consistency checks or dedicated hardware can be used for validity checking. Alternatively, a separate algorithm, using separate registers, could be run on the same processor to check the validity of a control action, with external time-out hardware being used to guarantee a response. These procedures could certainly provide a substantial cost savings over concurrent fault-detection methods.

For a system to be used in a general-purpose, time-sharing environment, the method of checking processors non-concurrently is very powerful because simple, relatively inexpensive schemes will suffice to guarantee the security of a user's environment. The price that is paid is to not detect some faults that could cause contamination of a user's own information. But conventional time-sharing systems have this handicap in addition to not having a high availability and not maintaining security in the presence of faults, so a clear improvement would be realized here at a fairly low cost. In order to detect failures as rapidly as possible in processors that have no concurrent fault-detection capability, periodic surveillance tests can be run which will determine if the processor is integral.

VALIDATION OF THE SOLITARY-FAULT ASSUMPTION

Fault-tolerant systems which are capable of isolating a faulty unit, and reconfiguring to run without it, typically can operate with several functional units removed at any given time. However, in order to design the system so that all possible types of failures can be handled, it is usually necessary to assume that at most one active unit is malfunctioning at any given time. The problem becomes essentially intractable when arbitrary combinations of multiple faults are considered. That is not to say that all cases of multiple faults will bring a system down, but usually no explicit effort is made to handle most multiple faults. Of course by multiple faults we mean multiple independent faults. If a failure of one unit can affect another, then the system must be designed to handle both units malfunctioning simultaneously or isolation must be added to limit the influence of the original fault.

A quantitative analysis will now be given which provides a basis for evaluating the viability of utilizing non-concurrent integrity-confirmation techniques in an adaptive fault-tolerant system. In the analysis below, the letter "s" will be used to designate the probability that two independent, simultaneous faults will cause a system to crash.

The next concept we need is that of coverage. Coverage is defined[5] as the conditional probability that a system will recover given that a failure has occurred. The letter "c" will be used to denote the coverage of a system.

In order to determine a system's ability to remain continuously available over a given period of time, it is necessary to know how frequently the components of the system are likely to fail. The usual measure employed here is the mean-time-between-failures. The letter "m" will be used to designate this parameter. It should be noted here that "m" represents the mean-time-between-internal-failures of a system; the system itself hopefully has a much better characteristic.

The final parameter that will be needed here is the maximum-time-to-recovery. This is defined to be the maximum time elapsed between the time an arbitrary fault occurs and the time the system has successfully reconfigured to run without the faulty unit. The letter "r" will be used to designate this parameter.

The commonly used assumption that a system does not deteriorate with age over its useful life will be adopted. Therefore, the exponential distribution will be used to characterize the failure probability of a system. Thus, at any given time, the probability of encountering a fault within the next u time units is:

$$\begin{aligned} p &= \int_0^u (1/m)*\exp(-t/m)\,dt \\ &= 1-\exp(-u/m) \end{aligned}$$

From this we can see that the upper bound on the conditional probability of encountering a second independent fault is given by:

$$q=1-\exp(-r/m)$$

Since it is obvious that r must be made much smaller than m if a system is to have a high probability of surviving many internal faults, the following approximation is quite valid:

$$\begin{aligned} q&=1-\exp(-r/m)\\ &=1-\sum_{k=0}^{\infty}(-r/m)^{k}/k!\\ &=1-1+r/m-(\tfrac{1}{2})*(r/m)^2+(\tfrac{1}{6})*(r/m)^3-\cdots\\ &\approx r/m \end{aligned}$$

Therefore, the probability of not being able to recover from an arbitrary internal failure is given by:

$$\begin{aligned} x&=(1-c)+c*q*s\\ &=(1-c)+c*s*r/m \end{aligned}$$

where the first term represents the probability of failing to recover due to a solitary failure and the second term represents the probability of not recovering due to simultaneous failures given that recovery from the first fault was possible.

If we now consider each failure as an independent Bernoulli trial and make the assumption that faulty units are repaired at a sufficient rate so that there is never a problem with having too many units logically removed from a system at any given time, then it is a simple matter to determine the probability of surviving a given period, T, without encountering a system crash. The hardware failures will be treated as n independent samples, each with probability of success $(1-x)$, where n is the smallest integer greater than or equal to T/m. Thus, the probability of not crashing on a given fault is $(1-x)=c*(1-r*s/m)$ and the probability, P, of not crashing during the period T is given by:

$$\begin{aligned} P&=[c*(1-r*s/m)]^n\\ &=c^n*(1-r*s/m)^n \end{aligned}$$

With this equation, it is now possible to establish the validity of using the various non-concurrent techniques mentioned above to confirm the integrity of a system. What this equation will establish is how often it will be necessary to perform the fault injection, action simulation, and surveillance procedures in order to gain an acceptable probability of no system crashes. Since the time required to detect, locate, and isolate a fault, and reconfigure to run without the faulty unit, will be primarily a function of the time to detection for the non-concurrent schemes and since this time is essentially equivalent to how frequently the confirmation procedures are invoked, we can assume that r is equal to the time period between the periodic integrity confirmation checks. In order to gain a feeling for the order of r, rather pessimistic numbers can be assumed for m, s, and T. Assume $m=1$ week, $s=\tfrac{1}{2}$, and $T=10$ years; this gives an n of 520. For now, assume c is equal to one. Now, in order to attain a probability of .95 that a system will survive 10 years with no crashes under the above assumptions, r will have to be:

$$\begin{aligned} r&=m/s*[1-.95^{(1/520)}]\\ &=119 \text{ seconds} \end{aligned}$$

Thus, if the periodic checks are made even as infrequently as every two minutes, a system will last 10 years with a probability of not crashing of approximately .95.

The effects of the coverage must now be examined. In order for the coverage to be good enough to provide a probability of .95 of no system crashes in 10 years due to the system's inability to handle single faults, it must be:

$$\begin{aligned} c&=.95^{(1/520)}\\ &=.9999 \end{aligned}$$

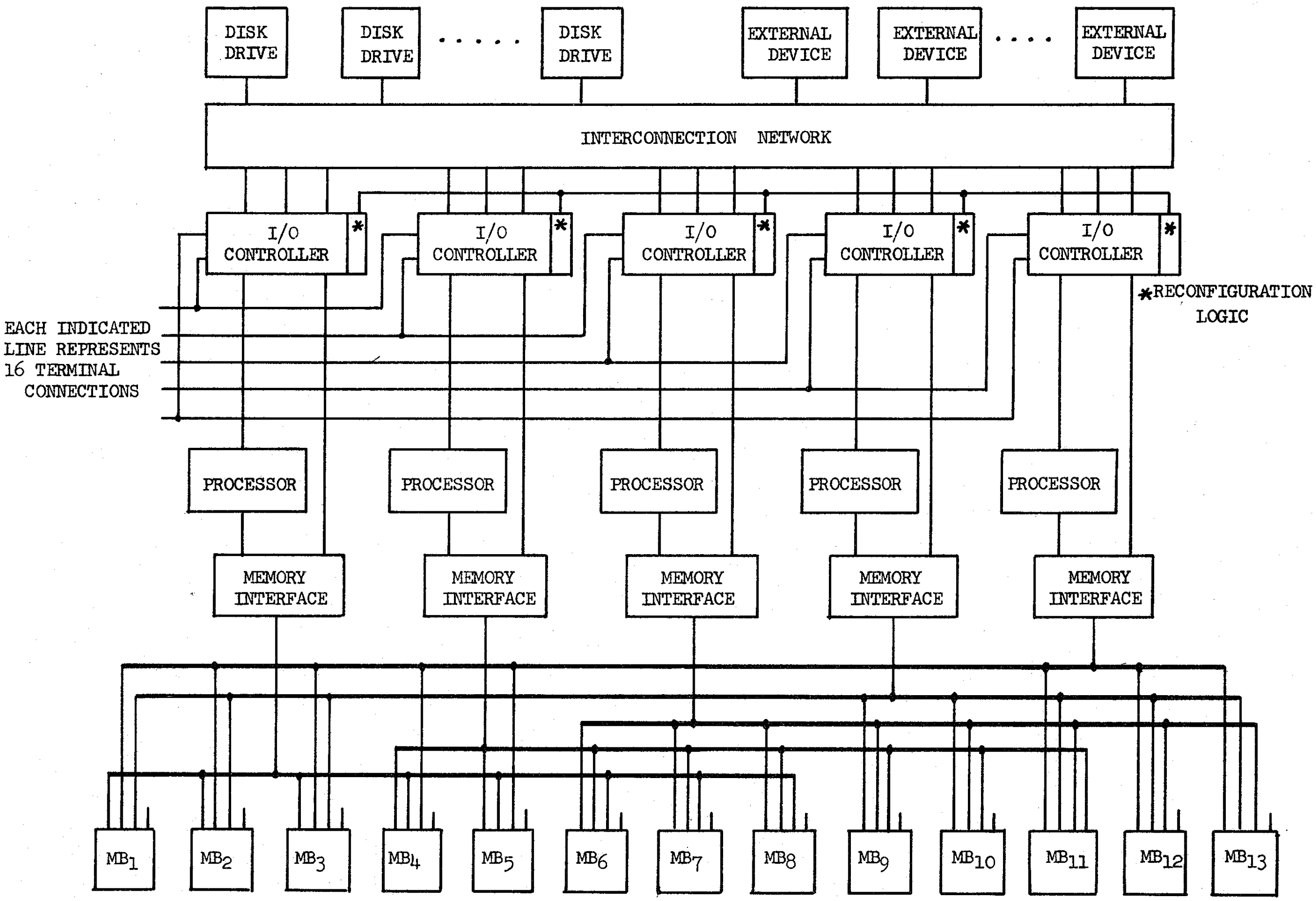
Now this would indeed be a very good coverage. Since the actual coverage of any given system will most likely fall quite short of this value, it seems that the coverage, and not multiple simultaneous faults, is the limiting factor in determining a system's ability to recover from faults.

The most important conclusion to be drawn from this section is that the solitary-fault assumption is not only convenient but quite justified, and this is true even when only periodic checks are made to verify the integrity of some of the logic.

INTEGRITY CONFIRMATION FEATURES OF THE "PRIME" SYSTEM

In order to better illustrate the potential power of dynamic integrity confirmation techniques, a description will now be given of how this concept is being used to economically provide an integrity confirmation structure for a fault-tolerant system.

At the University of California, Berkeley, we are currently building a modular computer system, which has been named PRIME, that is to be used in a multi-access, interactive environment. The initial version of this system will have five processors, 13 8K-word by 33-bit memory blocks with associated switching units,



EACH MEMORY BLOCK (MB) CONSISTS OF TWO 4K MODULES

Figure 1—Block diagram of the PRIME system

15 high-performance disk drives, and a switching network which allows processor, disk, and external-device switching. A block diagram of PRIME appears in Figure 1.

The processing elements in PRIME are 3-bus, 16-bit wide, 90*ns* cycle time microprogrammable processors called META 4*s*.[6] Each processor emulates a target machine in addition to performing I/O and executive functions directly in microcode. At any given time, one of the processors is designated the Control Processor (CP), while the others are Problem Processors (PPs). The CP runs the Central Control Monitor (CCM) which is responsible for scheduling, resource allocation, and interprocess message handling. The Problem Processors run user jobs and perform some system functions with the Extended Control Monitor (ECM) which is completely isolated from user processes. Associated with each PP is a private page, which the ECM uses to store data, and some target-machine code which it occasionally causes to be executed. A more complete description of the structure and functioning of PRIME is given elsewhere.[7]

The most interesting aspects of PRIME are in the areas of availability, efficiency, and security. PRIME will be able to withstand internal faults. The system has been designed to degrade gracefully in the presence of internal failures.[8] Also, interprocess integrity is always maintained even in the presence of either hardware or software faults.

The PRIME system is considered continuously integral if it is providing interprocess interference protection. Therefore, security must be maintained at all times. Other properties, such as providing user service and recovering from failures, can be handled in a less stringent manner. Thus, dynamic confirmation of system integrity in PRIME must be handled concurrently for interprocess interference protection and can be handled periodically with respect to the rest of the system. Of course, there are areas which do not affect interprocess interference protection but which will nonetheless utilize concurrent fault detection simply because it is expedient to do so.

Fault injection is being used to check most of the fault-detection logic in PRIME. This decision was made because the analysis of non-concurrent integrity-confirmation techniques has established that periodic fault injection is sufficiently effective to handle the job and because it is simpler and cheaper than the alternatives. There is a characteristic of the PRIME system that makes schemes which utilize periodic checking very attractive. At the end of each job step, the current process and the next process are overlap swapped. That is, two disk drives are used simultaneously; one of these disks is rolling the current job out, while the other is rolling the next job in. During this time, the associated processor has some potential free time. Therefore, this time can be effectively used to make whatever periodic checks may be necessary. And since the mean time between job steps will be less than a second, this provides very frequent, inexpensive periodic checking capabilities.

The integrity of Problem Processors is checked at the end of each job step. This check is initiated by the Control Processor which passes a one-word seed to the PP and expects the PP to compute a response. This seed will guarantee that different responses are required at different times so that the PP cannot accidently "memorize" the correct response. The computation requires the use of both target machine instructions and a dedicated firmware routine to compute the expected response. The combination of these two routines is called a surveillance procedure. This surveillance procedure checks all of the internal logic and the control storage of the microprocessors. The target machine code of the surveillance routine is always resident in the processor's private page. The microcode part is resident in control storage. A fixed amount of time is allowed for generating a response when the CP asks a PP to run a surveillance on itself. If the wrong response is given or if no response is given in the allotted time, then the PP is assumed to be malfunctioning and remedial action is initiated. In a similar manner, each PP periodically requests that the CP run a surveillance on itself. If a PP thinks it detects that the CP is malfunctioning, it will tell the CP this, and a reconfiguration will take place followed by diagnosis to locate the actual source of the detected error. More will be said later about the structure of the reconfiguration scheme.

While the periodic running of surveillance procedures is sufficient for most purposes, it does not suffice for protecting against interprocess interference. As previously mentioned, this protection must be continuous. Therefore, a special structure has been developed which is used to prevent interprocess interference on a continuous basis.[4] This structure provides double checks on all actions which could lead to interprocess interference. In particular, the validity of all memory and disk references, and all interprocess message transmissions, are among those actions double checked. A class code is used to associate each sector (1K words) of each disk pack with either a particular process or with the null process, which corresponds to unallocated space. A lock and key scheme is used to protect memory on a page (also 1K words) basis. In both cases, at most one process is bound to a 1K-word piece of physical storage. The Central Control Monitor is responsible for allocating each piece of storage, and it can allocate only those pieces which are currently unallocated. Each process is responsible for deallocating any piece of storage that it no longer needs. Both schemes rely on two processors and a small amount of dedicated hardware to provide the necessary protection against some process gaining access to another process' storage.

In order for the above security scheme to be extremely effective, it was decided to prohibit sharing of any storage. Therefore, the Interconnection Network is used to pass files which are to be shared. Files are sent as regular messages, with the owning process explicitly giving away any information that it wishes to share with any other process. All interprocess messages are sent by way of the CP. Thus, both the CCM and the destination ECM can make consistency checks to make sure that a message is delivered to the correct process.

The remaining area of integrity checking which needs to be discussed is the reaction hardware. In the PRIME system, this includes the isolation, power switching, diagnosis, and reconfiguration logic. A variety of schemes have been employed to confirm the integrity of this reaction logic. In order to describe the methods employed to confirm the integrity, it will be necessary to first outline the structure of the spontaneous reconfiguration scheme used in the PRIME system.

There are four steps involved in reorganizing the hardware structure of PRIME so that it can continue to operate with internal faults. The first step consists of detecting a fault. This is done by one of the many techniques outlined in this paper. In the second step, an initial reconfiguration is performed so that a new processor, one not involved in the detection, is given the job of being the CP. This provides a pseudo "hard core" which will be used to initiate gross diagnostics. The third step is used to locate the fault. This is done by having the new CP attach itself to the Programmable Control Panel[9] of a Problem Processor via the Interconnection Network, and test it by single-stepping this

PP through a set of diagnostics. If a PP is found to be functioning properly, then it is used to diagnose its own I/O channels. After the fault is located, the faulty functional-unit is isolated, and a second reconfiguration is performed to allow the system to run without this unit.

Of the four steps involved in responding to a fault, the initial reconfiguration poses the most difficulty. In order to guarantee that this initial reconfiguration could be initiated, a small amount of dedicated hardware was incorporated to facilitate this task. Associated with each processor is a flag which indicates when the processor is the CP. Also associated with each processor is a flag which is used to indicate that this processor thinks the CP is malfunctioning. For every processor, these two flags can be interrogated by any other processor. Each processor can set only its own flag that suggests the CP is sick. The flag which indicates that a processor is the CP can be set only if both the associated processor and the dedicated hardware concur. Thus, the dedicated hardware will not let this flag go up if another processor already has its up. Also, this flag will automatically be lowered whenever two processors claim that the CP is malfunctioning.

There is somewhat of a dilemma associated with confirming the integrity of this logic. Because of the distributed nature of this reconfiguration structure, it should be unnecessary to make any of it "reliable." That is, the structure is already distributed so that a failure of any part of it can be tolerated. However, if simultaneous faults are to be avoided, the integrity of this logic must be dynamically confirmed. Unfortunately, it is not practical to check this logic by frequently initiating reconfigurations. This dilemma is being solved by a scheme which partially simulates the various actions. The critical logic that cannot be checked during a simulated reconfiguration is duplicated so that infrequent checking by actual reconfiguration is sufficient to confirm the integrity of this logic.

The only logic used in the diagnostic scheme where integrity confirmation has not already been discussed is the Programmable Control Panel. This pseudo panel is used to allow the CP to perform all the functions normally available on a standard control panel. No explicit provision will be made for confirming the integrity of the Programmable Control Panel because its loss will never lead to a system crash. That is, failures in this unit can coexist with a failure anywhere else in the system without bringing the system down.

For powering and isolation purposes, there are only four different types of functional units in the PRIME system. The four functional units are the intelligence module, which consists of a processor, its I/O controller and the subunits that directly connect to the controller, its memory bus, and its reconfiguration logic; the memory block, which consists of two 4K-word by 33-bit MOS memory modules and a 4×2 switching matrix; the switching module, which consists of the switch part of two processor-end and three device-end nodes of the Interconnection Network; and the disk drive. The disk drives and switching modules can be powered up and down manually only. The intelligence modules must be powered up manually, but they can be powered down under program control. Finally, the memory blocks can be powered both up and down under program control.

No provision was made to power down the disks or switching modules under program control because there was no isolation problem with these units. Rather than providing very reliable isolation logic at the interfaces of the intelligence modules and memory blocks, it was decided to provide additional isolation by adding the logic which allows these units to be dynamically powered down. Also, because it may be necessary to power memory blocks down and then back up in order to determine which one has a bus tied up, the provision had to be made for performing the powering up of these units on a dynamic basis. Any processor can power down any memory block to which it is attached, so it was not deemed necessary to provide for any frequent confirmation of the integrity of this power-down logic. Also, every processor can be powered down by itself and one other processor. These two power-down paths are independent so again no provision was made to frequently confirm the integrity of this logic. In order to guarantee that the independent power-down paths do not eventually fail without this fact being known, these paths can be checked on an infrequent basis.

All of the different integrity confirmation techniques used in PRIME have been described. The essence of the concept of dynamic confirmation of system integrity is the systematic exploitation of the specific characteristics of a system to provide an adequate integrity confirmation structure which is in some sense minimal. For instance, the type of use and the distributed intelligence of PRIME were taken advantage of to provide a sufficient integrity-confirmation structure at a much lower cost and complexity than would have been possible if these factors were not carefully exploited.

REFERENCES

1 B BORGERSON C V RAVI
On addressing failures in memory systems
Proceedings of the 1972 ACM International Computing Symposium Venice Italy pp 40-47 April 1972

2 D A ANDERSON G METZE
Design of totally self-checking check circuits for M-out-of-N

codes
Digest of the 1972 International Symposium on
Fault-Tolerant Computing pp 30-34

3 R A SHORT
The attainment of reliable digital systems through the use of redundancy—A survey
IEEE Computer Group News Vol 2 pp 2-17 March 1968

4 R S FABRY
Dynamic verification of operating system decisions
Computer Systems Research Project Document No P-14.0
University of California Berkeley February 1972

5 W G BOURICIUS W C CARTER
P R SCHNIEDER
Reliability modeling techniques for self-repairing computer systems
Proceedings of the ACM National Conference pp 295-309
1969

6 *META 4 computer system microprogramming reference manual*
Publication No 7043MO Digital Scientific Corporation
San diego California June 1972

7 H B BASKIN B R BORGERSON R ROBERTS
PRIME—A modular architecture for terminal-oriented systems
Proceedings of the 1972 Spring Joint Computer Conference
pp 431-437

8 B R BORGERSON
A fail-softly system for time-sharing use
Digest of the 1972 International Symposium on
Fault-Tolerant Computing pp 89-93

9 G BAILLIU B R BORGERSON
A multipurpose processor-enhancement structure
Digest of the 1972 IEEE Computer Society Conference
San Francisco September 1972 pp 197-200

The in-house computer department

by JOHN J. PENDRAY

TECSI-SOFTWARE
Paris, France

INTRODUCTION

Over fifteen years ago, in some inner recess of some large corporation, a perplexed company official stood pondering before a large corporate organizational chart on his office wall. In his hand he held a small square of paper on which the words "Computer Department" were inscribed. Behold one of the modern frontiersmen of twentieth century business: the first man to try to stick the in-house computer department on the company organizational chart. He probably failed to find a place with which he felt comfortable, thereby becoming the first of many who have failed to resolve this problem.

Most of the earlier attempts ended by putting the computer department somewhere within the grasp of the corporate financial officer. The earliest computer applications were financial in nature, such as payroll, bookkeeping, and, after all, anything that costs as much as a computer must belong in the financial structure somehow. Many corporations are still trying to get these financial officers to recognize that there are many non-financial computer applications which are at least as important as the monthly corporate trial balances. Additionally, and perhaps even worse, the allocation of the computer department's resources is viewed as a relatively straightforward financial matter subject to budgeting within financial availability. This method of resource dispensing seems not to provide the right balance of performance and cost generally sought in the business world.

As the computer department growth pattern followed the precedent of Topsy, many corporations began to wonder why something that had become an integral part of every activity in the company should belong to one function, like finance. This questioning led to a blossoming forth of powerful in-house computer departments disguished under surcharged names like Information Services Department. Often, this square on the organizational chart had a direct line to the chief executive's office. This organizational form has created two of the most widely adopted erroneous concepts ever to permeate corporate activity. The first, and perhaps least damaging of these, is the concept that the highest corporate officers should be directly in touch with the computer at all times (and at any cost) to take advantage of something called the Management Information System (MIS). (Briefly, a MIS is a system designed to replace a fifteen-minute telephone call to the research department by a three-second response from a computer, usually providing answers exactly fourteen minutes and fifty-seven seconds faster than anyone can phrase his precise question.) The second concept to follow the attachment of the computer department to the chief executive's office has been the missionary work which has been undertaken in the name of, and with the power or influence of, the chief executive. Information service missionary work generally consists of the computer experts telling each department exactly what their information needs are and how they should go about their business.

This article will examine the nature of the in-house computer department in terms of its place in the corporate structure, its product, its function in the maturing of the product, and its methods of optimizing its resource utilization. Additionally, one possible internal structure for an in-house computer department will be presented.

THE IN-HOUSE COMPUTER DEPARTMENT WITHIN THE CORPORATE STRUCTURE

Most of the blocks on the corporate organizational chart have some direct participation in the business of the company. Take an example. The Whiz-Bang Corporation is the world's leader in the production of whiz-bangs. Its sales department sells whiz-bangs. Its production department produces whiz-bangs. Its development department develops new types of whiz-bangs. Its computer department has nothing to do with whiz-bangs. The people in the computer department know lots about computers, but their knowledge about whiz-bangs comes from what other departments have told them. What are they doing in the whiz-bang operation?

The computer department provides services to all the other departments in the company. These other departments are directly involved in the business of the company, but the function of the computer department is to provide services to the company, not to contribute directly in the business of the company. In this light, the computer department is like an external supplier of services.

How should such a supplier of services be funded? Let's return to the Whiz-Bang Corporation analogy. The marketing department is allocated sufficient resources to market whiz-bangs; the production department gets resources adequate to produce whiz-bangs, etc. It is not possible to allocate resources to the computer department on the basis of its direct contribution to whiz-bangs.

The computer department provides services, and these services are what should be funded. The value of these services provides the basis for funding. Other departments use the computer services, and it follows that only these departments can place the value on a service and that each department should pay for the services which it gets. Therefore, the funding of the computer department is the sum total of the payments received from the other departments for services rendered.

How should the computer department be controlled? First of all, it is necessary to define what is to be controlled. Either one can control product specifications or one can control the resources necessary to produce a product. Product specifications are generally controlled, in one way or another, by the buyer, while resource control is usually an internal problem concerned with the production of the buyer-specified product. At the Whiz-Bang Corporation, the marketing determines the buyer-desired product specifications, but each internal department calculates and controls its resource requirements to yield the specified number and type of whiz-bangs.

If the nature of the computer department is to provide services as its product, the users of these services should control their specifications. After all, they are paying for them (or should be).

If the computer department has the task of providing services that the other departments will be willing to fund, it should have the responsibility to allocate its resources to optimize its capability to provide the services. After all, they are the experts (or should be).

In resume, the departments in the corporation are using an external type of service from an internal source, the in-house computer department. Only they can value the service, but they won't do this job of valuation unless they are charged for the service. This valuation will automatically produce customer-oriented specifications for the services. On the other hand, once the services are specified and accepted at a certain cost, it is the job of the computer department to use its revenues in the best manner to produce its services. That is, the funding flows as revenues from the other departments; but the utilization of this funding is the proper responsibility of the provider of the services, the computer department.

These principles indicate that the in-house computer department can be melded into the corporate structure in any position where it can be equally responsive to all of the other departments while controlling, itself, the utilization of its resources.

THE PRODUCT OF THE COMPUTER DEPARTMENT—THE COMPUTER SERVICE

A computer service, which is the product produced and sold by the computer department, has an average life span of between five and ten years. It is to be expected that as the speed of computer technological change diminishes, this life span will lengthen. To date, many computer services have been conceived and developed without a real understanding of the nature of a computer service. The lengthening of the life span of the computer service should produce a more serious interest in understanding this nature in order to produce more economical and responsive services.

A well-conceived computer service is a highly tuned product which depends on the controlled maturing and merging of many technical facets. Too often this maturing and merging is poorly controlled because the life cycle of the computer service is not considered. The net result may be an inflexible and unresponsive product which lives on the edge of suicide, or murder, for the entirety of its operational life. Computer services management should not allow this inflexibility to exist, for the computer is one of the most flexible tools in the scientific grabbag. This innate flexibility should be exploited by management in the process of maturing a computer service.

MATURING THE COMPUTER SERVICE

There are four major phases in the maturing process: definition, development, operation, and overhaul. Perhaps the most misunderstood aspect of this maturing process is the relation between the phases in the life cycle of a computer service and the corresponding changes required in the application of technical specialties. Each phase requires a different technical outlook and a different level of technical skills.

The definition phase

Defining the service is oriented toward producing the functional specifications which satisfy the needs and constraints of the client. From another point of view, this is the marketing and sales problem for the computer department. It should be treated as a selling problem because the service orientation of the computer department is reinforced by recognition that the buyer has the problem, the money, and the buying decision.

The technical outlook should be broad and long term, for the entire life of the service must be considered. Technical details are not desirable at this stage, but it is necessary to have knowledge of recent technical advances which may be used to the benefit of the service. Also, a good understanding of the long-range direction and plans of the computer department is necessary in order to harmonize the proposed service with these goals.

The first step in defining a computer service is to locate the potential clients and estimate their susceptibility to an offer of a computer service. At first glance, this seems an easy task as the potential clients are well-known members of the corporate structure. Not so! Many of the most promising avenues of computer services cut across the normal functional separations and involve various mixtures of the corporate hierarchy. These mixtures are frequently immiscible, and the selling job involves convincing each participant of his benefit and helping him justify his contribution. The corporate higher-ups would also need to be convinced, but the money will seldom come from their operating budgets. In any case, the responsibility to seek out and sell new computer services lies with the computer department; however, the decision to buy is the sole property of the client departments.

After potential clients are identified, a complete understanding of the problem must be gained in order to close the sale. This understanding should give birth to several alternative computer system approaches giving different performance and cost tradeoffs. The potential customer will want to understand the parameters and options available to him in order to select his best buy. This is a phase of the life cycle of the service where the computer department provides information and alternatives to the prospective client.

Closing of the agreement should be in contractual terms with each party obligated for its part of the responsibility. All terms such as financing schedules, product specifications, development schedules, modification procedures, and penalties should be reduced to writing and accepted before work begins. A computer department that cannot (or will not) make firm commitments in advance of a project is poorly managed. (Of course there can always be a periodic corporate reckoning to insure that imbalances are corrected.)

The development phase

The contract is signed; the emphasis for the computer department changes from sales to development and implementation of the service. This phase calls for a concentrated, life-of-the-effort technical outlook with in-depth and competent technical ability required at all levels. The specialists of the computer department must be organized to produce the system which will provide the service as specified. The usual method for accomplishing this organization is the "project". Many learned texts exist on the care and feeding of a technical project, so let's examine here only the roles of the computer department and the client within the general framework of a project.

Computer department participation centers on its role as being the prime responsible party for the project. It is the computer department's responsibility to find the best techniques for satisfying all the goals of the project.

The correct utilization of the resources available to the computer department is a key to the project's success. One resource is time, and time runs out for a project. That is to say that no true project succeeds unless it phases out on time. A project team produces a product, turns it over to the production facility, and then the project ceases to exist.

The personnel resource of the computer department is also viewed differently in a project. The project team is composed of a hand-tailored mix of specialists who are given a temporary super-incentive and then removed from the project after their work is done. Super-incentives and fluid workforces are not easily arranged in all companies, and this is one of the reasons why the computer department must maintain control of the utilization of its resources.

The computer department should acquire new resources for a project within the following guideline: don't. Projects should not collect things around them or they become undisintegratable. The only exception: acquisitions which form part of the product, and not part of the project, and which will go with the product into the production phase.

Assuring the continuing health of the project's product is another critical aspect of the computer department's responsibility in the project. Since the project team will die, it must provide for the product to live independently of the project. This involves producing a turnoverable product which is comprehensible at all levels of detail. Also, the final product must be flexible enough to respond to the normal changes required during its lifetime.

It is interesting to note that in the development phase of the life cycle of a service, the project philosophy dictates that the computer department orient itself toward project goals and not just toward satisfying the specifications of the service. That is, the service specifications are only one of the project goals along with time, cost, etc.

On the other hand, the eventual user of the service, i.e., the client department, views the project as only a part of the total process necessary to have the service. To the client, the project is almost a "necessary evil"; however, the development project philosophy depends on active client involvement. Three distinct client functions are required. In their order of importance they are:

1. Countinuing decision-making on product performance and cost alternatives surfaced during the project work.
2. Providing aid to the technical specialists of the computer department to insure that the functional specifications are well understood.
3. Preparing for use of the service, including data preparation, personnel training, reorganization, etc.

These three client functions are certainly important aspects of a project, but it should not be forgotten that the development project is a method used by the computer department to marshal its resources and, therefore, must be under the responsibility of the computer department.

Development of the service may be an anxious phase as the client has been sold on the idea and is probably eager for his first product. This eagerness should not be blunted by the project team, nor should it affect the sound judgment of the team. Consequently, contact between the technical experts and the client should be controlled and directed toward constructive tasks.

The operation phase

The third step in the life cycle of a service begins when the development project begins to phase out. This is the day-to-day provision of the service to the client. In this phase, the computer department has a production philosophy which is single-minded: to assure the continuing viability of the service. This is often a fire-fighting function in which the quick-and-dirty answer is the best answer. There isn't much technical glory in this part of the life cycle of a service, but it's the part that produces the sustaining revenues for the computer department.

The computer department enhances continuing product viability by performing two functions. Of primary importance is to reliably provide the specified service with minimum expenditure of resources. Secondarily, the client must be kept aware of any possible operational changes which might affect the performance or cost of his service. Again, the client has a strong part in the decision to effect a change.

The client must contribute to the continuing viability of the product by using it intelligently and periodically evaluating its continuing worth.

The overhaul phase

As a service ages during its operational heyday, the environment around it changes little by little. Also, the quick-and-dirty maintenance performed by the operations personnel will begin to accumulate into a patchwork quilt which doesn't look much like the original edition. These two factors are not often self-correcting, but they can go unnoticed for years.

The only answer is a complete technical review and overhaul. Every service should be periodically dragged out of the inventory and given a scrub-down. This is another job where the technical glamor is quite limited; however, overhauling services to take advantage of new facilities or concepts can provide significant gains, not to mention that the service will remain neat, controllable, flexible, and predictable.

Thus definition, development, operation, and overhaul are the four phases in the life cycle of a computer service. All of these phases directly affect the clients and are accomplished with their aid and involvement. However, there is another area of responsibility for the computer department that does not touch the clients as closely. This area is the control over the utilization of the computer department's resources.

OPTIMIZING THE UTILIZATION OF THE COMPUTER DEPARTMENT'S RESOURCES

This important responsibility of the computer department is an internally-oriented function which is not directly related to the life cycles of the services. This is the problem of selecting the best mix of resources which fulfills the combined needs of the clients. In the computer service business there are two main resources, people and computing equipment.

Effective management of computer specialists involves at least training, challenging, and orienting. If these three aspects are performed well, a company has a better chance of keeping its experts, and keeping them contributing.

Training should be provided to increase the professional competence of the staff, but in a direction which is useful to the company. It is not clear, for instance, that companies who use standard off-the-shelf programming systems have a serious need to train the staff in the intricate design of programming systems software. It's been done, and every routine application suddenly became very sophisticated, delicate, and incomprehensible. However, training which is beneficial for the company should be made interesting for the personnel.

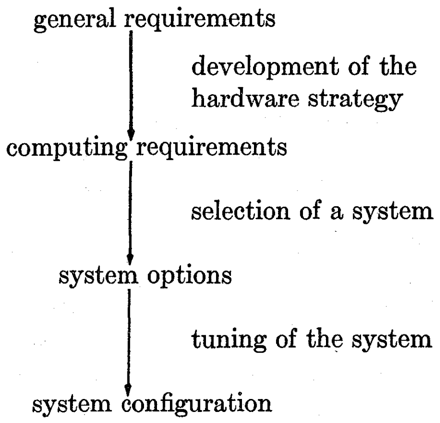
Challenging technical experts is a problem which is often aggravated by a poor hiring policy which selects over-qualified personnel. Such people could certainly accomplish the everyday tasks of the company if only they weren't so bored. The management problem of providing challenge is initially solved by hiring people who will be challenged by the work that exists at the company. Continuing challenge can be provided by increasing responsibility and rotating tasks.

Orienting the technical personnel is a critical part of managing the computer department. If left alone, most technical specialists tend to view the outside world as it relates to the parameter list of his logical input/output module, for example. He needs to be oriented to realize that his technical specialty is important because it contributes to the overall whole of the services provided to the clients. This client-oriented attitude is needed at all levels within a service organization.

Besides personnel, the other major resource to be optimized by the computer department is the computing system. This includes equipment and the basic programs delivered with the equipment, sometimes called "hardware" and "software".

Optimizing of a computing system is a frequently misunderstood or neglected function of the computer department. In a sense this is not surprising as there are three factors which obscure the recognition of the problem. First of all, computers tend to be configured by technical people who *like* computers. Secondly, most computer systems have produced adequate means of justifying themselves, even in an unoptimized state. Lastly, computer personnel, both manufacturers and users, have resisted attempts to subject their expenditures to rigorous analysis. It seems paradoxical that the same computer experts who have created effective analysis methodologies for so many other fields maintain that their field is not predictable and not susceptible to methodological optimization.

The utilization of computer systems is capable of being analyzed and may be seen as three distinct steps in the life cycle of the resource. These three steps can be presented diagrammatically as follows:



All too often, the strategy is chosen by default, the selection is made on the basis of sales effectiveness, and the tuning is something called "meeting the budget."

Development of the hardware strategy

Many computer departments don't even realize that different strategies exist for computing. This is not to say that they don't use a strategy; rather that they don't know it and haven't consciously selected a strategy.

The hardware strategy depends on having an understanding of the general needs of the computer department. The needs for security, reliability, independence, centralization of employees, type of computing to be done, amount of computing, etc., must be formulated in general terms before a strategy decision can be made. There are many possible ways to arrange computing equipment, and they each have advantages, disadvantages, and, as usual, different costs. The problem is to pick the strategy which responds to the aggregate of the general needs.

Perhaps some examples can best demonstrate the essence of a computing strategy. A large oil company having both significant scientific and business processing decides to separate the two applications onto two machines with each machine chosen for its performance/cost in one of the two specialized domains. A highly decentralized company installs one large economical general purpose computer but with remote terminals each of which is capable of performing significant independent processing when not being used as a terminal. A highly centralized company installs two large mirror-image general purpose computers with remote terminals which are efficient in teletransmission.

This is one area where the in-house computer department is not exactly like an external supplier of services, for the system strategy must reflect the general needs, and constraints, of the whole corporation.

Selection of a system

After the strategy is known, it becomes possible to better analyze and formulate the computing needs in terms of the chosen strategy. This usually results in a formal specification of computing requirements which includes workload projections for the expected life of the system. This is not a trivial task and will consume time, but the service rendered by the eventual system will directly depend on the quality of this task.

Once an anticipated workload is defined, one is free to utilize one, or a combination, of the methods commonly used for evaluating computer performance. Among these are simulation, benchmarks, and technical expert analysis.

One key decision will have a great influence on the results of the system selection: is a complete manufacturer demonstration to be required? This question should not be answered hastily; because a demonstration requires completely operational facilities, which may guarantee that the computer department will get yesterday's system, tomorrow. On the other hand, not having a demonstration requirement may bring tomorrow's most advanced system, but perhaps late and not quite as advanced as expected.

In any case, some methodology of system selection is required, if only to minimize the subjectivity which is so easily disguised behind technical jargon.

Tuning of the system

The winner of the hardware selection should not be allowed to start to take advantage of the computer department once the choice is made. On the contrary, the computer department is now in its strongest position as the parameters are much better defined.

One more iteration on the old specifications of requirements can now be made in light of the properties of the selected system. Also, an updating of the workload estimates is probably in order. Armed with this information, the computer department is now ready to do final battle to optimize the utilization of the system.

This optimization involves more than just configuring the hardware. It is a fine tuning of the computing environment. Take an example. As a result of the characteristics of the selected computer system, it might turn out that the mix of jobs "required" during the peak hours dictates that the expensive main memory be 50 percent larger than at any other time. Informing the clients of this fact, and that the additional memory cost will naturally be spread over their peak period jobs, will usually determine if all the requirements are really this valuable to the client. The client has the right to be informed of problems that will directly affect his service or costs. Only he can evaluate them and decide what is best for him.

Tuning of the environment involves selecting the best technical options, fully exploiting the potential of the computing configuration, and otherwise varying the parameters available. The trick is to examine all the parameters in the environment, not just the technical ones. This tuning process should be made, on a periodic basis, to insure that the environment remains as responsive as possible to the current needs.

PROPOSAL—AN ORGANIZATIONAL STRUCTURE FOR THE IN-HOUSE COMPUTER DEPARTMENT

It may not be possible to organize every computer department in the same manner, but some orientation should be found which would minimize the lateral dependencies in the organization. Perhaps a division of responsibilities based on the time perspective would be useful. Something as simple as a head office with three sections for long-range, medium-range, and short-range tasks could minimize lateral dependencies and still allow for exploitation of innate flexibility. In the language of the computer department, these sections might be called the head office, planning, projects, and operations, as shown in Figure 1.

The head office

There are three functions which must be performed by the head office. These functions are those which

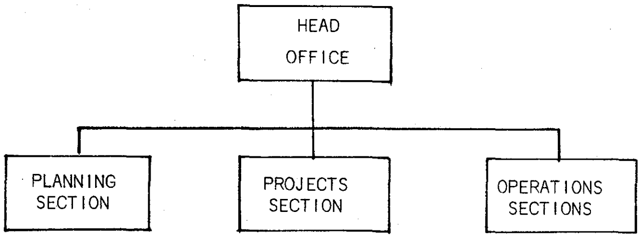


Figure 1

encompass all of the other sections and are integral to the computer department.

The first, and most important, of the functions for the head office is certainly marketing and client relations. All aspects of a service's life cycle involve the customer and he must be presented with a common point of contact on the business level. Every client should feel that he has the attention of city hall for resolving problems. In the opposite direction, the three sections should also use the head office for resolving conflicts or making decisions which affect the clients.

The second function of the head office is to control the life cycle of a service. As a service matures from definition to development to operations, it will be passed from one section to another. This phasing avoids requiring the technical people to change their outlook and skills to match the changes in the maturing process, but may create problems as a service is passed from hand to hand. Only the head office can control the process.

Resource control is the last function of the head office. The allocation of the various resources is an unavoidable responsibility and must reflect the changing requirements of the computer department.

The planning section

This is the long-range oriented group which must combine technical and market knowledge to plan for the future. The time orientation of this section will vary from company to company, but any task which can be considered as being in the planning phase is included.

Among the planning tasks is the development of long-range strategy. This strategy must be founded on a knowledge of expected customer needs (market research), advances in technical capabilities (state-of-the-art studies), and constraints on the computer department (corporate policy). Development of an equipment strategy is a good example of this task.

Another planning function is the developing of functional specifications for potential new services. In this respect, the planning section directly assists the head office in defining new services for clients.

Lastly, the planning section assists the projects section by providing state-of-the-art techniques which can be used in developing a specified service.

The projects section

This section has responsibility for the tasks in the computer department which are between planning and operation. Included is both development of services and changes in the technical facilities. The time orientation is limited for each task and each task is executed in a project approach.

A permanent nucleus of specialists exists to evaluate and implement major changes in the equipment. Each such major change is limited in its scope and accomplished on a project basis.

Development of services is naturally a task for the projects section. Each such project is performed by a team composed of people from the permanent nucleus and from the other two sections. The leadership comes from the projects section to insure that the project philosophy is respected, but utilization of personnel from the other sections assists in the transistions from planning to projects and from projects to operations.

This latter transition from development to operations is a part of the third function of the projects section. Direct aid is given to the operations section to insure that project results are properly understood and exploited in the day-to-day operations.

The operations section

Here is the factory. The time orientation is immediate. There are five major tasks to be performed, each of which is self-evident.

- Day-to-day production of the services,
- Accounting, analysis and control of production costs,
- Installation and acceptance of new facilities,
- Maintenance of all facilities (this includes systems software and client services),
- Recurring contact, training, and aid to the clients in use of the services.

TWO EXAMPLES

Perhaps the functioning of this organization can be demonstrated by an example from each of the two major areas of services and resources.

The life cycle of a service may begin either in the planning section (as a result of market research) or in the head office (as a result of sales efforts). In any case, the definition of the service is directed by the head office and performed by the planning section. Once the contract is signed, the responsibility passes to the projects section and the project team is built for the development effort. On the project team there will be at least one member from the planning section who is familiar with the definition of the service. The operations section also contributes personnel to facilitate the turnover at the end of the project. Other personnel are gathered from the permanent nucleus and the sections as needed. Each project member is transferred to the project section for the life of the project. The service is implemented, turned over to the operations section, and the project team is disbanded. Daily production and maintenance are performed by the operations section as is the periodic overhaul of the system. Each change of sections and all client contacts are under the control of the head office.

For resource utilization a close parallel exists. The head office again controls the life cycle. As an example, take the life cycle of a computer system. The planning section would develop a strategy of computing which would be approved by the head office. When the time arrived for changing the computer system, the projects section would define a project and combine several temporary members and permanent nucleus personnel to form the project team. A computer system selection would be made in line with the strategy of computing, and the system would be ordered. The operations section would be trained for the new system and accept it after satisfactory installation. Periodic tuning of the computer system would be done by permanent personnel in the projects section with the cooperation of the operations section. The flow of responsibility for these two examples is represented by Figure 2.

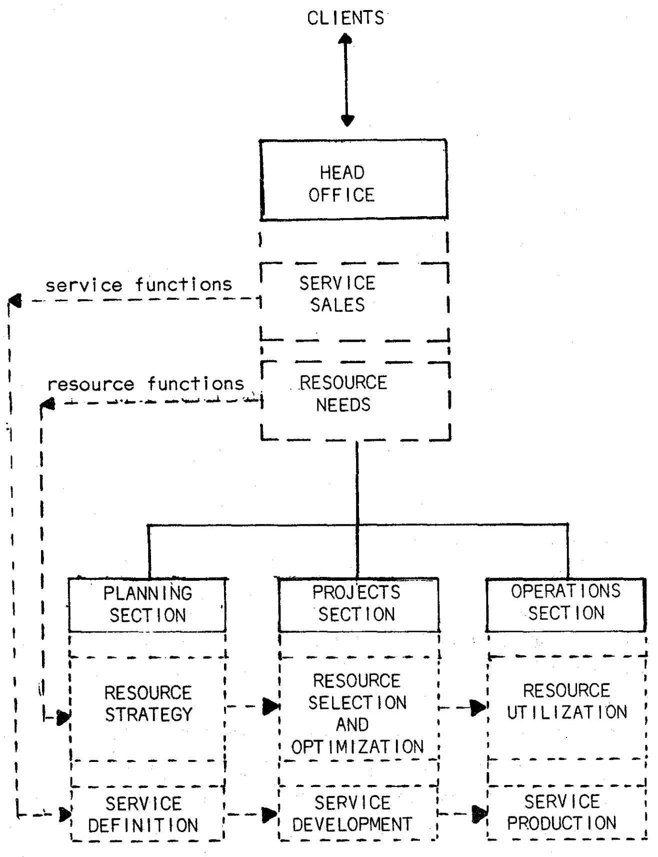


Figure 2

SUMMARY

Excepting those cases where the product of a company contains a computer component, the in-house computer department is in the business of providing an external service to the integral functions of a non-computer business. For this reason, the computer department does not appear to mesh well on an organizational chart of the departments which do directly contribute to the product line of the corporation. However, a well-founded in-house computer department which depends on its users for funds and on itself for the optimizing of the resources provided by these funds can peacefully serve within the organization.

The computer department can respond to these two principles of funding and resource control by recognizing that its funds depend on the satisfaction of the users and that the optimizing of the use of these funds can be aided by organizing around the life cycles of both the services provided and the resources used.

One possible organization designed to fulfill these two goals is composed of a head office and three sections. The head office maintains continuing control over the client relationship and over the life cycle of both services and resources. Each of the three sections specializes on a certain phase of the life cycle: definition, development, and operation.

Such an organizational approach for the computer department should provide:

- Computer services which are responsive to, and justified by, the needs of the users,
- A controlled and uniform evolution of the life cycle of both services and resources,
- A computer department management oriented towards dealing with technical considerations on a business basis,
- Technical personnel who are client-oriented specialists and who are constantly challenged and matured by dealing with different problems from different frames of reference,
- An in-house computer department which is self-supporting, self-evaluating, and justified solely by its indirect contributions to the total productivity of the corporate efforts.

A computer center accounting system

by F. T. GRAMPP

Bell Telephone Laboratories, Incorporated
Holmdel, New Jersey

INTRODUCTION

This paper describes a computer center accounting system presently in use at the Holmdel Laboratory and elsewhere within Bell Telephone Laboratories. It is not (as is IBM's SMF, for example), a tool which measures computer usage and produces "original" data from which cost-per-run and other such information can be derived. It is, rather, a collector of such data: it takes as input original run statistics, storage and service measurements from a variety of sources, converts these to charges, and reports these charges by the organizations (departments) and projects (cases) which incur them.

"DESIGN CRITERIA," below, outlines the overall functions of the system and describes the design criteria that must be imposed in order to assure that these functions can be easily and reliably performed.

The remainder of this paper is devoted to a somewhat detailed description of the data base (as seen by a user of the system) and to the actual implementation of the data base. Of particular interest is a rather unusual means of protecting the accounting data in the event of machine malfunction or grossly erroneous updates.

Finally, we describe backup procedures to be followed should such protection prove to be inadequate.

A description of the system interface is given in the Appendix for reference by those who would implement a similar system.

DESIGN CRITERIA

Many factors were considered in designing the system described here. The following were of major importance:

Cost reporting

Reporting costs is the primary function of any accounting system. Here, we were interested in accurate and timely reporting of charges by case (the term "case" is the accounting term we use for "project" or "account"), so that costs of computer usage to a project would be known, and by department, to ascertain the absolute and relative magnitude of computer expenses in each organization.

These orders of reporting are not necessarily identical, or even similar. For example, the cost of developing a particular family of integrated circuits might be charged against a single case, and computer charges for this development might be shared by departments specializing in computer technology, optics, solid state physics, and the like. Similarly, a single department may contribute charges against several or many cases—a good example of this is a drafting department.

Original charging information is associated with a *job number*, an arbitrary number assigned to a programmer or group of programmers, and associated with the department for which he works, and the project he is working on.

This job number is charged to one case and one department at any given point in time; however, the case and/or department to which it is charged may occasionally change, as is shown later.

Simplicity of modification

One thing that can be said of any accounting system is that once operational, it will be subjected to constant changes until the day it finally falls into disuse. This system is no exception. It is subjected to changes in input and output data types and formats, and to changes in the relationships among various parts of its data base. Response to such changes must be quick and simple.

Expansion capability

One of the more obvious unknowns in planning a system of this type is the size to which its data base may eventually grow. On a short term basis, this presents no problem: one simply allocates somewhat more storage than is currently needed, and reallocates periodically as excess space begins to dwindle. Two aspects of such a procedure must, however, be borne in mind: First, the reallocation process must not be disruptive to the day-to-day operation of the system. Second, there must be no reasonably foreseeable upper limit beyond which reallocation eventually cannot take place.

Protection

Loss of, say, hundreds of thousands of dollars worth of accounting information would at the very least be most embarrassing. Thus steps must be taken in the design of the system to guarantee insofar as is possible the protection of the data base. Causes of destruction can be expected to range from deliberate malfeasance (with which, happily, we need not be overly concerned), to program errors, hardware crashes, partial updating, or operational errors such as running the same day's data twice. If such dangers cannot be prevented, then facilities which recover from their effects must be available.

Continued maintenance

The most important design criterion, from the designer's point of view, is that the system be put together in such a way that its continued maintenance be simple and straightforward. The penalty for failure to observe this aspect is severe: the designer becomes the system's perpetual caretaker. On the other hand, such foresight is not altogether selfish when one considers the problems of a computer center whose sole accounting specialist has just been incapacitated.

THE DATA BASE: LEVEL 1

There are two ways in which to examine the data base associated with the accounting system. In the first case, there is its external appearance: the way it looks to the person who puts information into it or extracts information from it. Here, we are concerned with a collection of data structures, the way in which associations among the structures are represented, and the routines by means of which they are accessed. In the second, we look at its internal appearance: Here, we are interested in implementation details—in particular, those which make the system easily expansible and maintainable, and less vulnerable to disaster. These two aspects of the data base are, in fact, quite independent; moreover, to look at both simultaneously would be confusing. For this reason, we shall consider the first here, and defer discussion of the second to a later part of this paper. We first examine the structures themselves.

Tally records

Accounting system data is kept on disk in structures called tally records. Since we are concerned with data pertaining to cases, departments and job numbers, we have specified a corresponding set of tally records: Case Tally Records, Department Tally Records and Job Tally Records, respectively. These will be abbreviated as CTRs, DTRs and JTRs. In each tally record is kept the information appropriate to the particular category being represented. Such data fall naturally into three classes: fiscal information—money spent from the beginning of the year until the beginning of the present (fiscal) month; linkage data—pointers to associated records; other data—anything not falling into the other two categories.

For example, a CTR contains fiscal and linkage information: charges (a) up to and (b) for the current fiscal period, and a pointer to a chain of JTRs representing job numbers charged to the CTR's case.

A DTR's content is analogous to that of a CTR; the exception is the inclusion of some "other" data. When we report charges by case, the entire report is simply sent to the comptroller. Department reports, however, are sent to the heads of individual departments. To do so, we require the names of the department heads, and their company mailing addresses; hence the "other" data.

A JTR contains considerably more information: in addition to the usual fiscal and linkage information, a JTR contains pointers to associated case and department, data identifying the responsible programmer, and a detailed breakdown of how charges for the current month are being accumulated.

There is no way of determining *a priori* those things which will be charged for in order to recover computer center costs. In the olden days (say, 10 years ago) this was no problem: one simply paid for the amount of time he sat at the computer console. With today's computers, however, things just aren't that simple, since the computer center is called upon to provide all sorts of computing power, peripherals and services, and in turn, must recover the costs of said services from those who use them. Thus one might expect to find charges for CPU time, core usage, I/O, tape and disk storage rental, mounting of private volumes, telephone connect time, and so on. Add to this the fact that the charging algorithm changes from time to time, and it quickly becomes apparent that the number and kinds of charging categories simply defy advance specification.

Further, it seems clear that a given resource need not always be charged at the same rate—that in fact the rate charged for a resource should be a function of the way in which the resource is being used. For example, consider a program which reads a few thousand records from a tape and prints them. If such a program were to be run in a batch environment, in which printed output is first spooled to a disk and later sent to a high speed printer, one would expect the tape drive to be in use for only a matter of seconds. If the same program were to be run in a time-shared environment, in which each record read was immediately shipped to a teletype console for printing, the drive might be in use for several hours. If the computer center's charging algorithm is designed to amortize the rental costs of equipment among the users of the equipment, the latter use of "tape" ought to be considerably more expensive than the former, even though the same amount of "work" was done in each case.

For these reasons, we chose to make the process table-driven. In this way, new charging categories can be added, old ones deleted, and rates changed simply by editing the file on which a rate table resides. Such a scheme has the obvious drawback of requiring a table search for each transaction with the system, but the inefficiencies here are more than compensated by the ability to make sweeping changes in the charging information without having to reprogram the system.

Our rate table is encoded in such a way that it may be thought of as a two dimensional matrix. One dimension of the matrix consists of the *services* offerred by the computer center: batch processing (in our case, an ASP system), time shared services, data handling (a catch-all category which includes such things as tape copying, disk pack initialization and the like) storage rental, and sundry others. The other dimension consists of the usual computer *resources:* CPU time, core, disk and tape usage, telephone connect time, etc.

When a user incurs a charge, it is recorded in his JTR as a triple called a "chit." The chit consists of a service name, such as "ASP," a resource name, such as "CPU," and the dollar amount which he has been charged. In this implementation, each chit occupies twelve bytes:

	SERV	RES	COST
BYTE:	0	4	8

These chits are placed in an area at the end of the JTR. Initially, the area is empty. As time progresses and charges are accumulated, the number of chits in the JTR grows each time the job number is charged for a service-resource combination that it hasn't used before. The JTR itself is of variable length, and open-ended "to the right" to accommodate any number of chits that might be placed there.

Linkages

There are, in general, two ways in which one accesses information in the data base. Either one knows about a job number, and applies a charge against it and its associated case and department, or one knows about a case or department number and desires to look at the associated job numbers. This implies that there must be enough linkage information available for the following:

(a) Given a job number, find the case and department to which that number is charged.
(b) Given a case or department number, find all of the job numbers associated with that case or department.

The first case is trivial: one simply spells out, in a JTR, the case and department to which the job number is charged.

The second case is somewhat more interesting in that there may be one, or a few, or even very many job numbers associated with a single case or department. At Holmdel, we have the worst of all possible situations in this regard, in that the large majority of our cases and departments have very few job numbers associated with them, whereas a certain few have on the order of a hundred job numbers. Viewed in this light, schemes such as keeping an array of pointers in a CTR or DTR are, to say the least, unattractive because of storage management considerations.

What we have chosen to do, in keeping with our philosophy of open-endedness, is to treat the case-job and department-job structures as chains, and using the CTRs and DTRs as chain heads, operate on the chains using conventional list processing techniques. In our implementation, a case-job chain (more properly, the beginning of it) appears in a CTR as a character field containing a job number charged to that case. In the JTR associated with that job number, the chain is continued in a field which either contains another job number charged to the same case, or a string of zeros, which is used to indicate the end of a chain. Fields in the DTR and JTR function analogously to represent department job chains.

Traversing such a chain (as one frequently does while producing reports) is quite simple: begin at the beginning and get successive JTRs until you run out of pointers; then stop.

Inserting a new job number into a case- or department-job chain is also straightforward: copy the chain head into the chain field in the new JTR; then point the CTR or DTR to the new JTR. Deletion of JTRs from the system is accomplished by means of similar "pointer copying" techniques.

Indices

As was previously mentioned, the job numbers that are assigned to users are arbitrary. They happen, in point of fact, to be sequential for the most part, but this is simply a matter of clerical convenience. The only convention followed by case and department numbers is that (as of this writing) they are strictly numeric. This implies the necessity of a symbol table to associate names: case, department and job numbers, with their corresponding tally records on disk.

Three types of symbol table organization were considered for use with this system: sequential, in which a search is performed by examining consecutive entries; binary, in which an ordered table is searched by successively halving the table size; hash, in which a randomizing transformation is applied to the key. Of these, the sequential search is simply too slow to be tolerated. While the hashing method has a speed advantage over the binary method, the binary method has a very strong advantage for our application, namely, that the table is ordered.

One of the functions of the accounting system is that of producing reports, which are invariably ordered by case, department or job number. The ordering of the indices facilitates the work of many people who use the system.

In this implementation, there are three indices: one for cases, one for departments, and one for job numbers. These will be abbreviated CDX, DDX and JDX, respectively. Each index consists of some header information followed by pairs of names and pointers to associated tally records. Header information consists of five items:

RL: Record Length for the tally record. This is needed by the PL/I and OS/360 Input-Output routines.

TN: The number of entries currently in the index.

TMAX: The maximum number of entries which will fit in the core storage currently allocated the index.

R0,MRT: are not relevant at this time, and will be discussed later.

Entries are of the form (P1,P2,NAME), where P1 and P2 are 31-bit binary numbers pointing to records in a direct access data set, and NAME is a character string of appropriate length containing a case, job or department number.

Accessing techniques

Two types of access to the data base are required. The first is the *programmer's* access to the various structures and fields at the time he writes his program. The second is the *program's* access to the same information at the time the program is run.

The choice of PL/I as the language in which to write the system was, oddly enough, an easy one, since of all of the commonly available and commonly used languages for System/360, only PL/I and the assembler have a macro facility. Using assembly language would make much of the code less easily maintainable, and thus PL/I won by default.

The macro facility is used solely to describe various data base components to the routines that make up the accounting system by selectively including those components in routines which use them. Further, all references to these components are made via the macros. Adoption of this strategy has two somewhat related advantages: First,, it forces consistent naming of data items. Without the macros, one programmer would call a variable "X", another would call it "END-OF-MONTH-TOTAL", and so on. This, at least, would happen, and worse can be imagined. Second, should there be a change in a structure, all of the programs that use the structure must be recompiled. If the macros are used, the change can be made in exactly one place (the compile-time library) before recompilation.

Run-time access to the data base is achieved by following simple conventions and by using routines that have been supplied specifically for this purpose.

These conventions are simple because they are few and symmetric. The data base consists of six structures: the three indices, and the three types of tally records. None of these structures are internal to a program that interfaces with the data base. All of them are BASED, that is, located by PL/I POINTER VARIABLES which have been declared to be EXTERNAL so that they will be known to all routines in the system. Thus, for example, a program that accesses the JDX would contain the following declarations:

```
DCL 1 JDX BASED(PJDX),
                              /* The JDX is defined, an d */
% INCLUDE JDX;
                              /* its detailed description */
DCL PJDX POINTER EXTERNAL;
                              /* called from the library. */
```

The same convention applies to all of the other structures: they are allocated dynamically and based on external pointers whose names are the structure names prefixed by "P". A more detailed description of the user interface is given in the Appendix.

The foregoing implies that there is a certain amount of initialization work to be done by the system: setting pointers, filling indices and the like. This is, in fact, the case. Initialization is accomplished by calling a routine named INIT, usually at the start of the PL/I MAIN program. Among its other functions, INIT:

(a) Opens the accounting files. These include the six files containing the indices and tally records. Also opened are the file which contains the rate table, and a file used for JTR overflow.
(b) Allocates space for the indices and tally records, then sets pointers to the allocated areas.
(c) Reads into core the indices and the rate table, then closes these files. Some unblocking is required here both because the designers of PL/I (and indeed, of OS/360) have decreed that records shall not exceed 32,756 bytes in length, and because short records make the data base accessible to time shared programs running on our CPS system.

Once INIT returns control, the operating environment for the accounting system has been established. Indices are in core, and can be accessed by conventional programming techniques or by using the SEARCH, ENTER and DELETE routines, provided.

Reading and writing of tally records is also done by system routines, these being:

RDCTR	WRCTR
RDDTR	WRDTR
RDJTR	WRJTR

The read-write routines all require two arguments—a character string containing the name of the tally record to be read or written, and a logical variable which is set to signal success or failure to the caller. Actual data transfer takes place between a direct access data set and a based "TR" area in core. A typical example of the use of these routines is:

```
CALL RDJTR(MYJOB,OK); IF ¬ OK THEN STOP;
```

Two higher level routines, FORMJTR and LOSEJTR, are available for purposes of expanding or contracting the data base. FORMJTR examines the contents of JTR. If the JTR seems reasonable, that is, if it contains a case and department number, and its chain pointers are explicitly empty (set to zero) it performs the following functions:

(a) Checks to see if an appropriate CTR and DTR exist. If not, it creates them.
(b) Writes the JTR.
(c) Includes the JTR in the linkage chains extending from the CTR and DTR.

LOSEJTR performs exactly the inverse function, including deleting CTRs and DTRs whose chains have become empty as a result of the transaction.

INTERFACING WITH THE SYSTEM

Activities involving the system fall into four general categories: creating the data base, modifying the existing data base, inputting charges, and producing reports.

Creating the data base

No utility is provided in the system for the express purpose of creating the data base, because the form and format of previously extant accounting information varies widely from one Bell Laboratories installation to the next. A program has to be written for this purpose at each installation; however, the system routines provided are such that the writing of this program is a straightforward job requiring a few hours' work, at most. Briefly, creation of the data base proceeds as follows:

(a) Estimates are made of data set space requirements. These estimates are based on the number of cases, departments and job numbers to be handled, and on the direct access storage device capacities as described in IBM's *Data Management*[1] publication. Data sets of the proper size are allocated, and perhaps catalogued, using normal OS/360 procedures.
(b) An accounting system utility named RESET is

then run against the files. RESET initializes the indices so that they can communicate their core storage requirements to the system. No entries are made in the indices.

(c) The aforementioned installation-written routine is run. This routine consists of a two step loop: read in the information pertinent to a job number and construct a JTR; then call FORMJTR.

(d) At this point, the data base is installed. A program called UNLOAD is run so that a copy of the data base in its pristine form is available for backup purposes.

Modifying the data base

Two types of data base modifications are possible: those which involve linkage information, and those which do not. The latter case is most easily handled—an EDITOR is provided which accepts change requests in a format similar to PL/I's data directed input, then reads, modifies and rewrites a designated tally record. The former case is not so simple, however, and is broken down into several specific activities, each of which is handled by an accounting system utility supplied specifically for that purpose.

Authorizing new job numbers and closing old ones is done by a program called AUTHOR. This program adds a new entry to the data base by calling FORMJTR, and closes a job number by setting a "closed" bit to "1" in its JTR. Note that closed job numbers are not immediately deleted from the system.

Deleting closed job numbers is done once per year with an end-of-year program designed for that purpose. At this time, DTRs and CTRs which have no attached JTRs are also deleted from the system.

Changing the case or department number to which a job number is charged may be done in either of two ways. It is best to illustrate these by example.

In the first case, consider a department which has been renamed as a result of an internal reorganization. Its department number has been changed, say from 1234 to 5678, yet its work and personnel remain the same. In this case, it is desirable to delete "1234" from the DDX, install "5678", and change all "1234" references in the department-job chain to "5678".

As a second example, consider the case of a job number which was used by department 2345 but is now to be used by department 6789 due to a change in departmental work assignments. On the surface, this seems to be a matter of taking the job number out of 2345's chain and inserting it into 6789's. Unfortunately, it isn't that simple.

The charge fields in a chain, if added, should be equal to the field in the DTR at the chain head. Simply moving a JTR from one chain to another will make the old chain's fields sum low, and the new chain's fields sum high. The obvious solution to this problem is to forbid the changing of charged departments—i.e., to require that in the event that such a change is desired, the old job number be closed, and a new one authorized. Such a solution is not a very popular one, since job numbers have a habit of becoming imbedded in all sorts of hard-to-reach places—catalogued procedures, data set names and the like. Furthermore, it has been our experience that programmers develop a certain fondness for particular job numbers over a period of time and are somewhat reluctant to change them.

Our solution, then, is as follows: Given a job number, say 1234, whose charged department is to be reassigned, open a new job number, say 1234X, whose name was not previously known to the system, and which is charged to the proper department. Then close the old job number, and proceed to exchange names in the JTRs, and linkage pointers in the respective chains. A utility called SWAP is available which permits renaming or reassignment of either departments or cases (or both).

Inputting charges

As might be expected from our previous discussion of charging categories, there are many inputs to the accounting system. Moreover, the input formats are quite diverse, and subject to constant change. In order that the people charged with maintaining the accounting system might also be able to maintain their own sanity, it was necessary to design a simple way of incorporating new sources of charging information into the system.

Our first thought was to design a "general purpose input processor" i.e., a program that would read a data description and then proceed to process the data following (in this case, charge records). This approach was quickly abandoned for two reasons. First, the data description language required to process our existing forms of charge records would be quite complicated and thus difficult to learn and use, if in fact it could be implemented at all. Second, for each class of input charges, there is a certain amount of validity checking that can be performed at the time the charge records are read. Such checking need not be limited to a single record—for example, if it is known that a certain type of input consists of sequentially numbered cards, then a check can be made to determine whether some cards have been left out.

Our approach was as follows. For each type of charge record used by an installation, an input program must be written. This input program reads a charge record, does whatever checking is possible, constructs a standard structure consisting of a job number, service name, and one or more resource-quantity pairs, and passes this structure to a program called CHARGE.

CHARGE does the rest. It brings in the appropriate JTR, converts the quantities in the resource-quantity pairs to dollar charges via factors contained in the rate table, charges the JTR, adding chits if necessary, and charges the associated CTR and DTR. The important point here is that the writer of an input program is allowed complete freedom with respect to formats and checking procedures, while he is also allowed almost complete naivete with respect to the rest of the system.

Reporting

The system includes programs to produce three "standard" reports: one, (by cases) to be sent to the comptroller, one (by departments) to be sent to department heads, and a third (by job number) to be sent to the person responsible for each active job number in the system.

The comptroller's report is required of the computer center, and its format was specified in detail by the comptroller. The other two reports were designed to give the users of the computer center a clear and easily readable report of their computer usage in as concise a form as possible.

The department report shows old and recent charges to the department, followed by a list of job numbers being charged to that department. Accompanying each job number are its charges, the case to which it is charged, and the name of the person responsible for it. A more detailed breakdown is certainly possible; the average department head, however, usually doesn't want to see a breakdown unless something looks unusual. In that case, the programmer responsible for the unusual charges is probably his best source of information.

The user's report shows old and new charges for a job number, together with a detailed breakdown of the new charges by service-resource pairs. Its use to the programmer is threefold: it satisfies his curiousity—it enables him, in some cases, to detect and correct uneconomical practices—and it enables him to supply more detailed information to his department head should the need arise.

In order to produce the user's report, all of the chits in all of the JTRs in the system must be scanned. During the scanning process, it is a trivial matter to maintain a set of "grand totals" showing the money recovered by the computer center in terms of all service-resource categories. This valuable "by-product" is published after the user reports have been generated.

More specialized reporting is possible, but these programs, by their nature, are best written by particular installations rather than distributed as a part of the accounting system package. As was mentioned earlier, the ordering of the indices greatly facilitates the writing of such programs.

THE DATA BASE: LEVEL II

The foregoing discussion of the data base was aimed at the user of the system, and thus said nothing about its structure in terms of physical resources required, and the way in which these resources are used. We now expand on that discussion, concentrating on those factors influencing expansibility and protection. The main features of interest here are the implementation of tally record storage, the indices, and the provision to handle variable length JTRs.

Free storage pools

CTRs, DTRs and JTRs are stored on direct access data sets. When it is desired to access a tally record, a search of the appropriate index is made, and a relative record number on which the tally record is written is obtained from the index and used as the KEY in a PL/I read or write statement. The interesting feature of the system is that there is no *permanent* association between a particular tally record and a particular relative record number.

Direct access records used to contain tally records are stored in linked pools. The R0 entry in the appropriate index head points to the first available link, that link points to the second, and so on. One can think of the initial condition of a pool (no space used) as follows: R0 contains the number 1, record #1 contains the number 2, etc.

When a link is needed for tally record storage, a routine called GETLINK detaches the record pointed to by R0 from the free pool by copying that record's pointer into R0. The record thus detached is no longer available, and its number can be included into an index entry. A second routine called PUTLINK performs exactly the inverse function.

These activities are well hidden from the users of the system. The obvious advantage of the casual user not seeing the list processing operations is that he won't be confused by them. The disadvantage is that when he runs out of space and allocates a larger data set, he will forget to initialize the records by having the *nth* unused record point to the n+1st, as above. On the assumption (probably valid) that the data base will continue to grow over long periods of time, we have simplified the initialization procedure as follows:

(a) Let R0 point to the first available record (initially 1) and MRT point to the maximum record *ever* taken (initially 0).
(b) When GETLINK calls for a record, compare R0 and MRT. If R0>MRT then the data base is expanding into an area it never used before. In this case, set MRT=MRT+1. Otherwise, the record specified by R0 has been used in the past and has since been returned by PUTLINK, in which case we proceed as before.

With this procedure, initialization of the pools is done at the time that the data base is first created. Subsequent reallocation of data sets for purposes of enlarging the storage area is done as per standard OS/360 practice.

Indices and protection

Recall that although an index entry can be thought of as consisting of a pair (P,NAME) where P is a record number, and NAME is some character string, the entries are in fact represented as triples (P1,P2,NAME). At the time that an index is read into core by INIT, all of the P1s contain record numbers, while all of the P2s contain 0. Reading and writing of the tally records is done as follows. For reading:

(a) If the P2 entry is non-zero, read record # P2.
(b) Otherwise, read record #P1.

And for writing:

(a) If the P2 entry is zero, call GETLINK for a free record number, and copy it into P2.
(b) Write record # P2.

At the conclusion of a run in which the data base is to be updated, the main program, which had caused the operating environment to be established by calling INIT, now calls a routine named FINI, which in turn:

(a) Exchanges the P2 and P1 index entries in all cases where P2 is non-zero.
(b) Returns surplus links to the pool via PUTLINK.
(c) Rewrites the indices in more than one place.
(d) Closes all of the files.

Such a strategy offers both protection and convenience. Clearly, the danger of partial updating of the files during a charging run is minimized. Indeed, our standard operating instructions for those who run the system state that a job which crashes prior to completion is to be run a second time, unchanged. Further, a program that doesn't call FINI will not update the accounting files. Included in this category, besides "read only" reporting programs, are debugging runs, and input programs which contain algorithms to test the validity of incoming data, and which may not modify the files.

Variable length records

The JTRs, because of the fact that they can contain an unpredictable number of chits, are variable in length. Overflow records are obtained via GETLINK to extend the JTR as far as required. As read into storage by RDJTR, the overflow links are invisible to the user.

Besides the obvious convenience, the overflow handling in the JTR offers a different, if not devious type of protection. In the case of a system such as this, where the number of charging categories is, for practical purposes, unlimited, there is always the temptation to make the charging breakdown finer, and finer, and finer.

Succumbing to this temptation gives rise to nasty consequences. Processing time and storage space increase but the reports from the system become more voluminous, hence less readable, and in a sense contain less information because of the imprecision inherent in so many of the "computer usage measurement" techniques. (In this latter case, we often tend to behave analogously to the freshman physics student who measures the edges of a cube with a meter stick and then reports its volume to the nearest cubic millimicron.)

By happy coincidence, it turns out that in a system with "normal" charging categories, most JTRs have relatively few chits—too few to cause overflow—while occasional JTRs require one or more overflow records. Should the breakdown become fine enough that *most* of the JTRs cause overflow, the cost of running the accounting system rises—not gradually, but almost as a step. Further, if the breakdown is subsequently made coarser, the excess chits, and hence the overflow records, quietly disappear at the end of the next accounting period. Thus the system is, in a sense, forgiving, and tends to protect the user from himself.

BACKUP

As Mr. Peachum[2] aptly remarked, it has never been very noticeable that what ought to happen is what happens. In addition to our efforts to make the system crash-proof, we have also provided several levels of backup procedures.

Backup indices

As noted previously, FINI rewrites the indices, but in more than one place. Since the "extra" copies are written first, these can be copied to the "real" index files in the event that a crash occurs while the latter are being rewritten by FINI.

Unload-reload copies

Two utilities, UNLOAD and RELOAD, are supplied with the system. UNLOAD copies the structured files onto tape in such a way that the structure is preserved if RELOAD copies them back. It is our present practice to take UNLOAD snapshots daily, cycling the tapes once per week, and, with a different set of tapes, monthly, cycling the tapes once per year.

Since chits are deleted at the end of each month (for economy of storage) UNLOAD-style dumps are also useful if it becomes necessary to backtrack for any reason to a point in time prior to the beginning of the current month. Further, the tapes are in such a format that they are easily transmitted via data link to another installation for purposes of inspection or off-site processing.

OS/360 dump-restore

It is the practice, in our computer center, to periodically dump all of our permanently mounted direct access storage devices using the OS/360 Dump-Restore utility. Since the accounting files are permanently mounted, this procedure provides an additional level of safety.

Reformatting

The worst possible mishap is one in which the chains in the system, for one cause or another, are destroyed to the extent that one or more of them "points to the wrong place". Although this condition is most unusual, it is also most insidious, since there is a possibility that errors of this type can remain hidden for, perhaps, as long as a few weeks. If enough input data has been added to the data base to make it undesirable to backtrack to the point prior to that at which the initial error is suspected to have occurred, symbolic information sufficient to regenerate the pointers is contained in the data base, and routines have been provided to copy the data base, sans structure, onto a sequential file, and then to rebuild it, using FORMJTR.

ACKNOWLEDGMENT

The design and implementation of the accounting system described here was completed with the help and cooperation of many people, and for this the author is truly grateful. In particular, the efforts, advice, insight and inspiration provided throughout the project by Messers. R. E. Drummond and R. L. Wexelblat assured its successful completion.

REFERENCES

1 *IBM system/360 operating system data management services*
Order Form GC26-3746

2 B BRECHT
Die Dreigroschenoper

APPENDIX

The user-system interface

The facilities provided to give the user convenient access to the data base and the routines which manipulate it can be divided into two categories: compile-time facilities and run-time facilities.

Compile-time facilities

These consist of PL/I macro definitions describing various structures. Since the storage class of a structure (e.g., BASED, STATIC, etc.) may be different in different routines, or, where there are multiple copies of a structure, even within the same routine, the initial "DCL 1 name class," must be provided by the user.

Compile-time structures include the indices (CDX, DDX, JDX) the tally records (CTR, DTR, JTR) and the rate table.

Example 1:

```
          DCL 1 CDX BASED(PCDX),
              % INCLUDE CDX;
```

produces:

```
DCL 1 CDX BASED(PCDX),
  2 RL FIXED(15) BINARY,  /* Record Length */
  2 TN FIXED(15) BINARY,    /* # of Entries */
  2 TMAX FIXED(31) BINARY,
                                /* Max. Entries. */
  2 RO FIXED(31) BINARY,        /* Pool Head */
  2 MRT FIXED(31) BINARY,
                           /* Max. Record Taken */
  2 VAREA(0:N REFER(CDX.TN)),
                                /* Index Proper */
    3 P1 FIXED(31) BINARY,       /* Read Ptr. */
    3 P2 FIXED(31) BINARY,      /* Write Ptr. */
    3 NAME CHAR(9);           /* Case Number */
```

Example 2:

```
          DCL 1 CTR BASED(PCTR),
              % INCLUDE CTR;
```

produces:

```
DCL 1 CTR BASED(PCTR),
  2 CLNK FIXED(31) BINARY,
                          /* Used by GETLINK. */
  2 CCAS CHAR(9),            /* Case charged. */
  2 CUNUSED CHAR(3),        /* For future use. */
  2 COLD FIXED(31) BINARY,
                           /* $ to last fiscal. */
  2 CNEW FIXED(31) BINARY,
                             /* Latest charges. */
  2 CCUM FIXED(31) BINARY,
                             /* Cumulative total. */
  2 CJCH CHAR(8);   /* Job chain for this case. */
```

Example 3:

Since it is expected that the user will always use the system-supplied rate table (as opposed to a private copy of same):

```
               % INCLUDE RATES;
```

produces:

```
DCL 1 RATES BASED(PRTS),      /* Rate Data */
  2 #SERVICES FIXED BIN(31),
  2 TOT__RES FIXED BIN(31),
                              /* Tot. # Resources */
  2 SERVICE(12),            /* Classes of Service */
    3 NAME CHAR(4),
    3 CODE CHAR(1),       /* Comptroller's Code */
    3 #RESOURCES FIXED BIN(31),
    3 OFFSET
          /* Into Res. Table */ FIXED BIN(31),
  2 RES__TABLE(120),                /* Resources */
    3 NAME CHAR(20),
    3 ABBR CHAR(4),
    3 UNIT CHAR(8),
    3 RATE FLOAT DED(14);           /* Per-unit */
```

Run-time facilities

Routines are provided to establish and terminate the system's run-time environment, maintain the indices, fetch and replace tally records, expand and contract the data base, and handle allocation of disk storage. These are shown in Table I, below.

TABLE I—User Interface Routines

ROUTINE	FUNCTION	ARGUMENTS REQUIRED	EXAMPLE

INIT
FINI
	Initialization & termination.	None.	CALL INIT;
			CALL FINI;

SEARCH
ENTER
DELETE
	Index maintenance.	Index name, key name, return pointer, success indicator.	CALL SEARCH(DDX, '1234', RP, OK);
			IF ¬ OK THEN STOP;

RDCTR
RDDTR
RDJTR
WRCTR
WRDTR
WRJTR
	Read and write routines for tally records	Name (i.e. case, department or job number), success indicator.	CALL RDJTR('MYJOB', OK);
			IF ¬OK THEN DO;
			PUT LIST(MYJOB| |'MISSING');
			STOP;
			END;

FORMJTR
LOSEJTR
	Installation & deletion of job nos.	Job number, success indicator.	CALL FORMJTR(NEWJOB,OK);
			IF ¬OK THEN STOP;

GETLINK PUTLINK	Allocate & return disk space.	Data set name, pointer to 1st avail. record, return pointer.	CALL GETLINK (FILE,RP,POOLHD);

An approach to job pricing in a multi-programming environment

by CHARLES B. KREITZBERG and JESSE H. WEBB

Educational Testing Service
Princeton, New Jersey

INTRODUCTION

Computers are amazingly fast, amazingly accurate, and amazingly expensive. This last attribute, expense, is one which must be considered by those who would utilize the speed and accuracy of computers. In order to equitably distribute the expense of computing among the various users, it is essential that the computer installation management be able to accurately assess the costs of processing a specific job. Knowing job costs is also important for efficiency studies, hardware planning, and workload evaluation as well as for billing purposes.

For a second generation computer installation, job billing was a relatively simple task; since in this environment, any job that was in execution in the machine had the total machine assigned to it for the entire period of execution. As a result, the billing algorithm could be based simply upon the elapsed time for the job and the cost of the machine being used. In most cases, the cost for a job was given simply as the product of the run time and the rate per unit time. While this algorithm was a very simple one, it nevertheless was an equitable one and in most cases a reproducible one.

Because of the fact that in a second generation computer only one job could be resident and in execution at one time, the very fast CPUs were often under utilized. As the CPUs were designed to be even faster, the degree of under utilization of them increased dramatically. Consequently, a major goal of third generation operating systems was to optimize the utilization of the CPU by allowing multiple jobs to be resident concurrently so that when any one job was in a wait state, the CPU could then be allocated to some other job that could make use of it. While multiprogramming enabled a higher utilization of the CPU, it also introduced new problems in job billing. No longer was the old simple algorithm sufficient to equitably charge for the running of jobs. The two major reasons for this are:

- The sharing of resources by the resident jobs, and
- The variation in elapsed time from run to run of a given job.

Unlike the second generation computer a given job no longer has all of the resources that are available on the computer allocated to it. In a multi-programming computer, a job will be allocated only those resources that it requests in order to run. Additional resources, that are available on the computer, can be allocated to other jobs. Therefore, it is evident that the rate per unit time cannot be a constant for all jobs, as it was for second generation computer billing, but must in some sense be dependent upon the extent to which resources are allocated to the jobs.

The second item, and perhaps the most well-known, that influences the design of a billing algorithm for a third generation computer is the variation that is often experienced in the elapsed time from run to run of a given job. The elapsed time for any given job is no longer a function only of that job, but is also a function of the job mix. In other words, the elapsed time for a job will vary depending upon the kinds and numbers of different jobs which are resident with it when it is run.

In order to demonstrate the magnitude of variation that can be experienced with subsequent runs of a given job, one job was run five different times in various job mixes. The elapsed time varied from 288 seconds to 1,022 seconds. This is not an unusual case, but represents exactly what can happen to the elapsed time when running jobs in a multi-programming environment. The effect, of course, is exaggerated as the degree of multiprogramming increases.

Not only can this variation in run time cause a difference in the cost of a job from one run to another, but it also can cause an inequitability in the cost of different jobs; the variation in run time can effectively cause one job to be more expensive than another even though the amount of work being done is less.

Objectives

We have isolated several important criteria to be met by a multi-programming billing algorithm. Briefly, these criteria are as follows.

- *Reproducibility*—As our previous discussion has indicated, the billing on elapsed time does not provide for reproducibility of charges. Any algorithm that is designed to be used in a multiprogramming environment should have as a characteristic, the ability to produce reproducible charges for any given job regardless of when or how it is run, or what jobs it is sharing with the computer.
- *Equitability*—Any billing algorithm designed for use in a multi-programming environment must produce equitable costs. The cost of a given job must be a function only of the work that the job does, and of the amount of resources that it uses. Jobs which use more resources or do more work must pay more money. The billing algorithm must accommodate this fact.
- *Cost Recovery*—In many computer operations it is necessary to recover the cost of the operation from the users of the hardware. The billing algorithm developed for a multi-programming environment must enable the recovery of costs to be achieved.
- *Auditability*—A multi-programming billing algorithm must produce auditable costs. This is particularly true when billing outside users for the use of computer hardware. The charges to the client must be auditable.
- *Encourage Efficient Use of the Hardware*—Since one goal in a design of the third generation hardware was to optimize the use of that hardware, a billing algorithm that is designed for use in a multijobbing environment should be such that it encourages the efficient use of the hardware.
- *Allow for Cost Estimating*—The implementation of potential computer applications is often decided upon by making cost estimates of the expense of running the proposed application. Consequently, it is important that the billing algorithm used to charge customers for the use of the hardware also enables potential customers to estimate beforehand, the expense that they will incur when running their application on the computer hardware.

We distinguish between *job cost* and *job price: job cost* is the amount which it costs the installation to process a given job; *job price* is the amount that a user of the computer facility pays for having his job processed. Ideally, the job price will be based on the job cost but this may not always be the case. In many organizations, notably universities, the computer charges are absorbed by the institution as overhead; in these installations the job price is effectively zero—the job costs are not. In other organizations, such as service bureaus, the job price may be adjusted to attract clients and may not accurately reflect the job cost. In either case, however, it is important that the installation management know how much it costs to process a specific job.[1,2]

The development of the job billing algorithm (JBA) discussed in this paper will proceed as follows: first, we will discuss the "traditional" costing formula used in second generation computer systems:

$$\text{cost} = (\text{program run time}) \times (\text{rate per unit time})$$

and we shall demonstrate its inadequacy in a multijobbing environment. Second, we shall develop a cost formula in which a job is considered to run on a dedicated computer (which is, in fact, a subset of the multi-programming computer) in a time interval developed from the active time of the program.

DEVELOPMENT OF THE JOB PRICING ALGORITHM

In order to recover the cost of a sharable facility over a group of users, the price, P, of performing some operation requiring usage t is:

$$P = (C)\left(\frac{(t_k)}{\sum t_i}\right) \qquad (1)$$

where: C is the total cost of the facility

$\sum t_i$ is the total usage experienced
t_k is the amount of use required for the operation

Consider the billing technique which was used by many computer installations running a *single thread* (one program at a time) system. Let $\$m$ be the cost per unit time of the computer configuration. Then, if a program began execution at time t_1 and terminated execution at time t_2, the cost of running the program was computed by:

$$P = \$m(t_2 - t_1) \qquad (2)$$

As the utilization of the computer increased the cost per unit time decreased.

The cost figure produced by (2) is in many ways a very satisfying one. It is simple to compute, it is reproducible since a program normally requires a fixed time for its execution, it is equitable since a "large" job will cost more than a "small" job (where size is measured by the amount of time that the computers resources are held by the job). Unfortunately for the user, however, the cost produced by (2) charges for all the resources of the computer system even if they are unused.

This "inflated" charge is a result of the fact that, in a single thread environment, all resources of the computer system are allocated to a program being processed even if that program has no need of them. The effect of this is that the most efficient program in a single thread environment is the program which executes in the least amount of time; that is, programmers attempt to minimize the quantity (t_2-t_1); this quantity, called the *wall clock time* (WCT) of the program, determines the program's cost.

Since the rate of the computer is constant, the only way to minimize the cost for a given program is to reduce its WCT; in effect, make it run faster. Hence, many of the techniques which were utilized during the second generation were designed to minimize the time that a program remained resident in the computer.

The purpose of running in a multi-thread environment, one in which more than the one program is resident concurrently, is to maximize the utilization of the computer's resources thus reducing the unit cost.

In a multi-thread processing system, the cost formula given by (2) is no longer useful because:

1. It is unreasonable to charge the user for the entire computer since the unused resources are available to other programs.
2. The wall clock time of a program is no longer a constant quantity but becomes a function of the operating environment and job mix.

For these reasons we must abandon (2) as a reasonable costing formula. Many pricing algorithms are in use; however, none is as "nice" as (2). If possible, we should like to retain formula (2) for its simplicity and intuitive appeal.[3] This may be done if we can find more consistent definitions to replace m (rate) and WCT (elapsed time).

Computed elapsed time

A computer program is a realization of some process on a particular hardware configuration. That is, a program uses some subset of the available resources and "tailors" them to perform a specific task. The program is loaded into the computer's memory at time t_1 and terminates at time t_2. During the period of residency, the program may be in either one of two states: *active* or *blocked.* A program is *active* when it is executing or when it awaiting the completion of some external event. A program is *blocked* when it is waiting for some resource which is unavailable. These categories are exhaustive; if a program is not active and is not waiting for something to happen then it is not doing anything at all. The two categories are not, however, mutually exclusive since a program may be processing but also awaiting the completion of an event (for example an input/output operation) indeed, it is this condition which we attempt to maximize via channel overlap. Therefore, we define *voluntary wait* as that interval during which a program has ceased computing and is awaiting the completion of an external event. We define *involuntary wait* as the interval during which a program is blocked; a condition caused by *contention.*

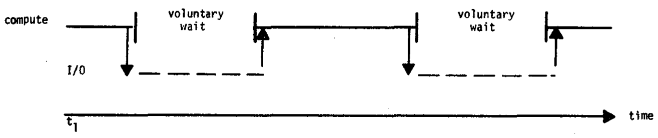


Figure 1—States of a program in a single thread environment

In general, voluntary wait results from initiation of an input/output operation and in a single thread system we have:

$$WCT = t_2 - t_1 = \Sigma t_c + \Sigma t_v \qquad (3)$$

where:

each t_c is a compute interval
and each t_v is a voluntary wait interval.

graphically, the situation is represented as in Figure 1. The solid line represents periods of compute and the broken line indicates intervals of input/output activity. Since Σt_c is based on the number of instructions executed which is constant and Σt_v is based on the speed of input/output which is also constant (except for a few special cases), WCT is itself constant for a given program and data in a single thread environment. The ideal case in this type of system is one in which the overlap is so good that the program obtains the $i+1$st record as soon as it has finished processing the ith

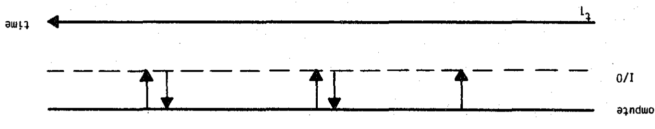


Figure 2—A program with maximum overlap

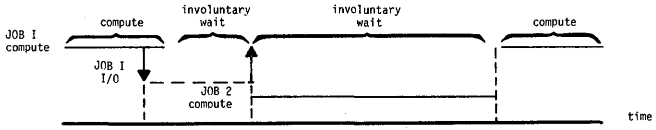


Figure 3—States of a program in a multi-thread environment

record. Graphically, this situation is shown in Figure 2 and we can derive the lower bound on WCT as:

$$\text{WCT} \geq \Sigma t_c \qquad (4)$$

and, of course:

$$\text{WCT} \rightarrow \Sigma t_c \quad \text{as} \quad \Sigma t_v \rightarrow 0 \qquad (5)$$

In a multi-thread environment, we know that:

$$\text{WCT} = \Sigma t_c + \Sigma t_v + \Sigma t_i \qquad (6)$$

where: t_i is an interval of involuntary wait.

But, from the above discussion we know that $\Sigma t_c + \Sigma t_v$ is a constant for a given program, hence, the inconsistency in the WCT must come from t_i. This is precisely what our intuition tells us; that the residency time of a job will increase with the workload on the computer. Graphically, a program running in a multi-thread environment might appear as in Figure 3.

During the interval that a program is in involuntary wait, it is performing no actions (in fact, some programmers refer to a program in this state as "asleep"). As a consequence, we may "remove" the segments of time that the program is asleep from the graph for time does not exist to a program activity in involuntary wait. This permits us to construct a series of time sequences for the various programs resident in the computer; counting clock ticks only when the program is active. When we do this a graph such as Figure 4 becomes continuous with respect to the program (Figure 5).

Of course, the units on the x-axis in Figure 5 no longer represents *real-time*, they represent, instead, the *active time* of the program. We shall call the computed time interval *computed elapsed time* (CET) defined as:

$$\text{CET} = \Sigma t_c + \Sigma t_v = \text{WCT} - \Sigma t_i \qquad (7)$$

and as $t_i \rightarrow 0$, CET→WCT so that we have the relationship:

$$\text{WCT} \geq \text{CET} \qquad (8)$$

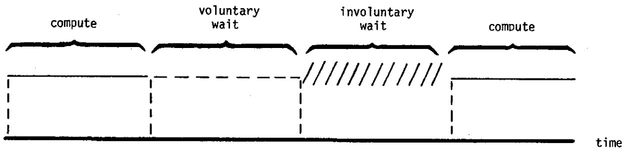


Figure 4—States of a program based on real time

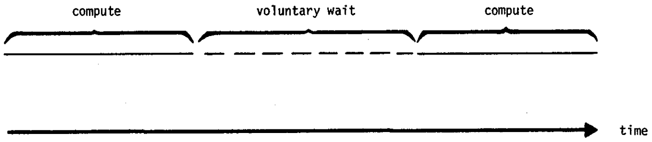


Figure 5—States of a program based on active time

The quantity WCT−CET represents the interference from other jobs in the system and may be used as a measure of the degree of multi-programming.

Unfortunately, the CET suffers from the same deficiency as the WCT—it is not reproducible. The reason for this is that on a movable head direct access storage device contention exists for the head and the time for an access varies with the job mix. However, the CET may be estimated from its parameters. Recall that $\text{CET} = \Sigma t_c + \Sigma t_v$. The quantity Σt_c is computed from the number of instructions executed by the program and is an extremely stable parameter. The quantity Σt_v is based upon the number and type of accesses and is estimated as:

$$\Sigma t_v \approx a(i) n_i \qquad (9)$$

where $a(i)$ is a function which estimates the access time to the ith file and n_i is the number of access to that file. The amount of time which a program waits for an input or output operation depends upon a number of factors. The time required to read a record is based upon the transfer rate of the input/output device, the number of bytes transferred, the latency time associated with the device (such as disk rotation, tape inter-record gap time, and disk arm movement). For example, a tape access on a device with a transfer rate of R_T and a start-stop time of S_T would require:

$$a(T) = S_T + R_T b \qquad (10)$$

seconds to transfer a record of b bytes. Hence, for a file of n records, we have a total input/output time of:

$$\sum_{i=1}^{n} (S_T + R_T b_i) = nS_T + R_T \Sigma b_i \qquad (11)$$

where Σb_i is the total number of bytes transferred. In practice $\Sigma b_i \approx nB$ where n is the number of records and B is the average blocksize. The term S_T is, nominally, the start-stop time of the device. However, this term is also used to apply a correction to the theoretical record time. The reason is that while the CET will never be greater than the I/O time plus the CPU time, overlap may cause it to be less. This problem is mitigated by the fact that at most computer shops (certainly at ETS) almost all programs are written in high-level computer languages and, as a result, the job mix is homogeneous. A measure of overlap may be obtained by fitting various curves to historical data and choosing the one which provides the best fit. In other words, pick the constants which provide the best estimate of the WCT.

It is important to remember that the CET function produces a *time* as its result. We are using program parameters such as accesses, CPU cycles, and tape mounts only because they enable us to predict the CET with a high degree of accuracy.

The original billing formula (2) which we wished to adapt to a multi-thread environment utilized a time multiplied by a dollar rate per unit time. The CET estimating function has provided us with a pseudo run time; we must now develop an appropriate dollar rate function.

In order to develop a charging rate function we consider the set of resources assigned to a program. In a multi-programming environment, the computer's resources are assigned to various programs at a given time. The resources are partitioned into subset computers each assigned to a program. The configuration of the subset computers is dynamic; therefore, the cost of a job is:

$$\text{cost} = \sum_{i=1}^{n} \text{CET}_i \cdot r_i \qquad (12)$$

where i is the allocation interval; that is, the interval between changes in the *job's* resources held by the job. CET_i is the CET accumulated during the ith interval. r_i is the rate charged for the subset computer with the configuration held by the program during interval i.

The allocation interval for OS/360 is a *job step.*

The rate function

Some of the attributes which the charging rate function should have are:

- the rate for a subset computer should reflect the "size" of the subset computer allocated; a "large" computer should cost more than a "small" computer.
- the rate for a subset computer should include a correction factor based upon the probability that a job will be available to utilize the remaining resources.
- the sum of the charges over a given period must equal the monies to be recovered, for that period.

With these points in mind, we may create a rate function.

The elements of the resource pool may be classified as *sharable resources* and *nonsharable resources.* Tape drives, core memory, and unit record equipment are examples of nonsharable resources; disk units are an example of a sharable resource. While these categories are not always exact they are useful since we assume that allocation of a nonsharable resource is more significant than allocation of a sharable resource. At Educational Testing Service, it was determined that the most used nonsharable resources are core storage and tape drives. Therefore, it was logical to partition the computer into subset computers based upon the program's requirement for core and tapes. Tapes are allocated in increments of one; core is allocated in 2K blocks. Hence, there are (# tapes * available core/2,000) possible partitions.

For any given design partition, we would like to develop a rate which is proportional to the load which allocation places upon the resources pool. A single job may sometimes disable the entire computer. If, for example, a single program is using all of the available core storage then the unused devices are not available to any other program and should be charged for. On the other hand, if a single job is using all available tapes, other jobs may still be processed and the charge should be proportionately less.

The design proportion is the mechanism by which the total machine is effectively partitioned into submachines based upon the resources allocated to the submachines. A design proportion can be then assigned to any job based upon the resources it requires. The design proportion should have at least the following properties.

- The design proportion should range between the limits 0 and 1.
- The design proportion should reflect the proportion of the total resources that are allocated to the job.
- The design proportion should reflect, in some fashion, the proportion of the total resources that are not allocated to the job, but which the job prevents other jobs from using.

The design proportion proposed for the billing algorithm is based upon the probability that when the job is resident, some other job can still be run. The definition of this parameter is as stated below.

> *The design proportion of a job is equal to the probability that when the job is resident, another job will be encountered such that there are insufficient resources remaining to run it.*

Since OS/360 allocates core in 2K blocks, the number of ways that programs can be written to occupy available core is equal to:

$$N = C/2 \qquad (13)$$

where,

N = Number of ways that programs can be written
C = Core available in Kilo-bytes

In addition, if there are T tapes available on the hardware configuration then there are T plus 1 different ways that programs can be written to utilize tapes. Therefore, the total number of ways that programs can be written to utilize core and tapes is given by the following equation,

$$N = (C/2)(T+1) \tag{14}$$

where,

N = Total number of ways that programs can be written
C = Core available in Kilo-bytes
T = Number of tape drives available

The design proportion for a given job can be alternately defined as 1 minus "the probability that another job can be written to fit in the remaining resources." This is shown as follows.

$$D_P = 1.0 - \frac{[(C_A - C_U)/2](T_A - T_U + 1)}{[(C_A)/2](T_A + 1)} \tag{15}$$

where,

D_P = Design proportion for the job
C_A = Core available in Kilo-bytes
C_U = Core used by job
T_A = Tape drives available on the computer
T_U = Tape drives used by the job

It is important to note that the sum of the design proportions of all jobs resident at one time can be greater than 1.0. For example, consider the following two jobs resident in a 10K, four tape machine.

Job #1: 6K; 1 Tape $D_P = 17/25$
Job #2: 4K; 3 Tapes $D_P = 19/25$

The sum of their design proportion is 36/25. This seems odd at first since the design proportion of a 10K; four tape job is 1.0. However, this can be shown to be a necessary and desirable property of the design proportion. To show that this is the case, it is necessary to consider the amount of work done and the total cost of the work for two or more jobs that use the total machine compared to the cost of the same amount of work done by a single job that uses the total machine. This analysis will not be covered here.

The design proportion function as defined herein is a theoretical function. It is based solely upon the theoretical possibility of finding jobs to occupy available resources. Clearly, the theoretical probability and the actual probability may be somewhat different. Consequently, a design proportion could be designed based upon the actual probabilities experienced in a particular installation. Such a probability function would change as the nature of the program library changed. The design proportion function described above would change only as the configuration of the hardware changed. Either technique is acceptable and the design proportion has the desired properties. That is, the design proportion increases as the resources used by the various jobs increase. However, it also reflects the resources that are denied to other jobs because of some one jobs' residency. Consider the fact that when all of core is used by a job, the tape drives are denied to other jobs. The design proportion in this case is 1.0 reflecting the fact that the job in effect has tied up all available resources even though they are not all used by the job itself.

While the design proportion function is simple, it has many desirable properties:

- It is continuous with respect to OS/360 allocation; all allocation partitions are available.
- It always moves in the right direction, that is, increasing the core requirement or tape requirement of the program, results in an increased proportion.
- It results in a proportion which may be multiplied by the rate for the total configuration to produce a dollar cost for the subset computer.
- It is simple to compute.

If it were determined that the required recovery could be obtained if the rate for the computer were set at $35 per CET minute, the price of a step is determined by the equation:

$$P_{\text{step}} = ((\$35.)D_P(\text{core, tapes}))\quad(\text{CET}/60) \tag{16}$$

and the price of a job (with n steps) is:

$$P_{\text{job}} = \sum_{i=1}^{n} P_{\text{step}} \tag{17}$$

We have come full circle and returned to our "second generation" billing formula:

$$\text{cost} = \text{rate} \cdot \text{time}$$

The key points in the development were:

- A multi-tasking computer system may be considered to be a collection of parallel processors by altering the time reference.
- The variation in time of a program run in a multi-

programmed environment is due to involuntary wait time.

- The computed elapsed time may be multiplied by a rate assigned to the subset computer and an equitable and reproducible cost developed.

IMPLEMENTATION OF THE JOB PRICING ALGORITHM

The Job Pricing Algorithm (JPA) is implemented under OS/360 Release 19.6. No changes to the operating system were required; a relatively minor modification was made to HASP in order to write accounting records to disk for inclusion in the accounting system. The basis of the JPA is the IBM machine accounting facility known as Systems Management Facility (SMF).[4]

Billing under the JPA involves four steps:

1. Collect the job activity records at execution time. The records are produced by SMF and HASP and are written to a disk data set—SYS1.MANX.
2. Daily, the SYS1.MANX records are consolidated into job description records and converted to a fixed format.
3. The output from step (2) is used as input to a daily billing program which computes a cost for the jobs and prepares a detailed report of the day's activity by account number.
4. Monthly, the input to the daily program is consolidated and used as input to a monthly billing program which interfaces with the ETS accounting system.

The raw SMF data which is produced as a result of job execution contains much valuable information about system performance and computer workload which is of interest to computer center management.

One useful characteristic of the JPA is that costs are predictable. This enables a programmer or systems analyst to determine, in advance, the costs of running a particular job and, more importantly, to design his program in the most economical manner possible. In order to facilitate this process, a terminal oriented, interactive, cost estimating program has been developed. This program is written in BASIC and enables the programmer to input various parameters of his program (such as filesize, CPU requirements, blocking factors, memory requirements) and the cost estimate program produces the cost of the program being developed. Parameters may then be selectively altered and the effects determined.

CONCLUSION

The approach to user billing described in this paper has proved useful to management as well as users. Many improvements are possible especially in the area of more accurate CET estimation. Hopefully, designers of operating systems will, in the future, include sophisticated statistics gathering routines as part of their product thus providing reliable, accurate data for accounting.

APPENDIX

A method of deriving CET parameters

Let the wall clock time (W) be estimated as follows,

$$W' = A_T X_T + A_D X_D + A_M X_M + C \qquad (1)$$

where,

$X_T = \#$ of tape accesses
$X_D = \#$ of disk accesses
$X_M = \#$ of tape mounts
$C = $ CPU time
$A_T, A_D, A_M = $ Coefficients to be determined

We wish to determine the coefficients A_T, A_D, and A_M that will maximize the correlation between W', the computed elapsed time, and W, the actual elapsed time. Define the error e as,

$$e = (W - W') \qquad (2)$$

$$\therefore e_i = W_i - A_T X_{T_i} - A_D X_{D_i} - A_M X_{M_i} - C_i \qquad (3)$$

The correlation coefficient, r, can be written as,

$$r^2 = 1 - (\sigma_e^2 / \sigma_w^2) \qquad (4)$$

Then, in order to maximize r^2, it is sufficient to minimize σ_e^2 since σ_w^2 is a constant over a given sample.

$$\sigma_e^2 = \Sigma(e - \bar{e})^2 = \Sigma e^2 - 2\Sigma e\bar{e} + \Sigma \bar{e}^2 \qquad (5)$$

Since

$$\Sigma \bar{e}^2 = n\bar{e}^2 \qquad (6)$$

we have,

$$\sigma_e^2 = \Sigma e^2 - \frac{2}{n}(\Sigma e)^2 + (\Sigma e/n)^2 \qquad (7)$$

$$\sigma_e^2 = \Sigma e^2 - n^{-1}(\Sigma e)^2 \qquad (8)$$

Finally, we have,

$$\sigma_e^2 = \Sigma[(W_i - C_i) - A_T X_T - A_D X_D - A_M X_M]^2 - n^{-1}[\Sigma(W_i - C_i)$$

$$-A_T\Sigma X_T-A_D\Sigma X_D-A_M\Sigma X_M]^2 \tag{9}$$

$$\begin{aligned}\frac{\partial\sigma_e^2}{\partial A_T}&=-2\Sigma[(W_i-C_i)-A_TX_T\\&-A_DX_D-A_MX_M]X_T+\frac{2}{n}[\Sigma(W_i-C_i)\\&-A_T\Sigma X_T-A_D\Sigma X_D-A_M\Sigma X_M]\Sigma X_T\end{aligned} \tag{10}$$

$$\begin{aligned}\frac{\partial\sigma_e^2}{\partial A_D}&=-2\Sigma[(W_i-C_i)-A_TX_T\\&-A_DX_D-A_MX_M]X_D+\frac{2}{n}[\Sigma(W_i-C_i)\\&-A_T\Sigma X_T-A_D\Sigma X_D-A_M\Sigma X_M]\Sigma X_D\end{aligned} \tag{11}$$

$$\begin{aligned}\frac{\partial\sigma_e^2}{\partial A_M}&=-2\Sigma[(W_i-C_i)-A_TX_T\\&-A_DX_D-A_MX_M]X_M+\frac{2}{n}[\Sigma(W_i-C_i)\\&-A_T\Sigma X_T-A_D\Sigma X_D-A_M\Sigma X_M]\Sigma X_M\end{aligned} \tag{12}$$

Since all the partials must vanish, we have,

$$\begin{aligned}&A_T\left[\Sigma X_T^2-\frac{(\Sigma X_T)^2}{n}\right]+A_D\left[\Sigma X_TX_D-\frac{\Sigma X_D\Sigma X_T}{n}\right]\\&\quad+A_M\left[\Sigma X_TX_M-\frac{\Sigma X_T\Sigma X_M}{n}\right]\\&\quad=\Sigma(W_i-C_i)X_T-\frac{\Sigma(W_i-C_i)\Sigma X_T}{n}\end{aligned} \tag{13}$$

$$\begin{aligned}&A_T\left[\Sigma X_TX_D-\frac{\Sigma X_T\Sigma X_D}{n}\right]+A_D\left[\Sigma X_D^2-\frac{(\Sigma X_D)^2}{n}\right]\\&\quad+A_M\left[\Sigma X_DX_M-\frac{\Sigma X_D\Sigma X_M}{n}\right]\\&\quad=\Sigma(W_i-C_i)X_D-\frac{\Sigma(W_i-C_i)\Sigma X_D}{n}\end{aligned} \tag{14}$$

$$\begin{aligned}&A_T\left[\Sigma X_TX_M-\frac{\Sigma X_T\Sigma X_M}{n}\right]+A_D\left[\Sigma X_DX_M-\frac{\Sigma X_D\Sigma X_M}{n}\right]\\&\quad+A_M\left[\Sigma X_M^2-\frac{(\Sigma X_M)^2}{n}\right]\\&\quad=\Sigma(W_i-C_i)X_M-\frac{\Sigma(W_i-C_i)\Sigma X_M}{n}\end{aligned} \tag{15}$$

Solving the simultaneous equations (13), (14), and (15) for A_T, A_D, and A_M should give values for the parameters that will maximize the correlation between the computed elapsed time and the actual elapsed time.

The technique was applied to a sample month of data which was composed of 19401 job steps. The coefficients determined were,

$A_T=$ 0.0251 seconds
$A_D=$ 0.0474 seconds
$A_M=81.2$ seconds

When these coefficients were used in Equation (1) to determine the computed elapsed time, the correlation coefficient between the computed time and actual time over the 19401 steps was 0.825. When other coefficients were used, i.e. $A_T=0.015$, $A_D=0.10$, and $A_M=60.0$, the correlation was only 0.71.

Note: Card read, card punch, and print time constants were not computed in this fashion simply because there is insufficient data on job steps that use these devices as dedicated devices. However, as data become available in the future, the method could be applied to obtain good access times.

REFERENCES

1 L L SELWIN
Computer resource accounting in a time sharing environment
Proceedings of the Fall Joint Computer Conference 1970

2 C R SYMONS
A cost accounting formula for multi-programming computers
The Computer Journal Vol 14 No 1 1971

3 J T HOOTMAN
The pricing dilemma
Datamation Vol 15 No 8 1969

4 IBM Corp
IBM System/360 operating system: System management facilities
Manual GC28-6712 1971

Facilities management—A marriage of porcupines

by DAVID C. JUNG

Quantum Science Corporation
Palo Alto, California

FM—DEFINED

FM definition often elusive

There are almost as many definitions for Facilities Management (FM) as there are people trying to define it. Because FM can offer different levels of service, some variations in its definition are legitimate.

FM was initiated by the Federal Government in the 1950's when the Atomic Energy Commission, the National Aeronautics and Space Administration (NASA), and the Department of Defense offered several EDP companies the opportunity to manage and operate some of their EDP installations. Previously, these companies had developed strong relationships with the various agencies through systems development and software contracts.

FM definition expanding

Nurtured by the Federal Government, FM has emerged as a legitimate computer service in the commercial EDP environment. Since FM has been offered in the commercial market, its definition has expanded to include additional services. In fact, customers are now beginning to expect FM vendors to have expertise that extends far beyond the day-to-day management of the data processing department.

Electronic Data Systems (EDS), formed in the early 1960's, pioneered the FM concept in the commercial market. Shortly after its founding, EDS recognized the massive EDP changes required in the hospital and medical insurance industry as a result of Medicare and other coverages changed by the Social Security Administration. Accordingly, EDS secured several State Blue Cross/Blue Shield organizations as customers. While operating these installations, EDS developed standard software packages that met the recordkeeping requirements of the Social Security Administration. Moreover, this software succeeded in improving operator control and reducing operating costs. Consequently, EDS marketed these software packages to other Blue Cross/Blue Shield organizations.

Outside of the medical insurance field, EDS has successfully pursued FM opportunities in life insurance, banking, and brokerage.

The success of EDS, both in revenue/profit growth and in the stock market did not go unnoticed by others in the computer services industry. As a result, in the late 1960's and early 1970's many software firms and data service bureaus diversified into FM—many, unfortunately, with no real capabilities. Since FM has proven itself as a viable business in the commercial market, over 50 independent FM firms have been formed. Moreover, at least 50 U.S. corporations with large, widespread computer facilities have spun off profit centers or separate corporations from their EDP operations. In many cases, these spinoffs offer customers FM as one of their computer services.

An ideal concept of FM

The ideal role for the FM vendor is to assist in all the tasks related to business information and the EDP operations in the firm. The Facilities Manager could assume full responsibility for the EDP operations, from acquiring the equipment and staffing the installation—to distributing the information to the firm's operating areas. FM also has a vital role in defining business information requirements for top management. More specifically, the FM vendor should be able to define what information is required to operate the business, based on his industry experience. He should also be able to help establish cost parameters, based on an analysis of what other firms in the industry spend for EDP. Moreover, FM vendors will assist top management to cost optimize the array of business processing methods which may include manual or semi-automated

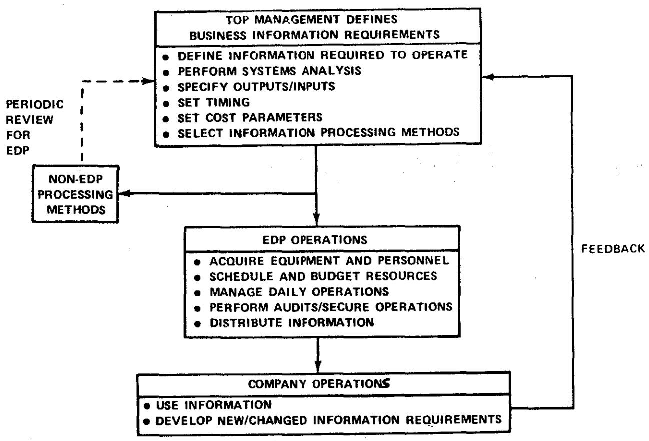

TOP MANAGEMENT DEFINES BUSINESS INFORMATION REQUIREMENTS
• DEFINE INFORMATION REQUIRED TO OPERATE
• PERFORM SYSTEMS ANALYSIS
• SPECIFY OUTPUTS/INPUTS
• SET TIMING
• SET COST PARAMETERS
• SELECT INFORMATION PROCESSING METHODS
PERIODIC REVIEW FOR EDP
NON-EDP PROCESSING METHODS
EDP OPERATIONS
• ACQUIRE EQUIPMENT AND PERSONNEL
• SCHEDULE AND BUDGET RESOURCES
• MANAGE DAILY OPERATIONS
• PERFORM AUDITS/SECURE OPERATIONS
• DISTRIBUTE INFORMATION
FEEDBACK
COMPANY OPERATIONS
• USE INFORMATION
• DEVELOP NEW/CHANGED INFORMATION REQUIREMENTS

Figure 1—Business information and EDP in the ideal firm

approaches as well as EDP. The FM vendor must be skilled at working with personnel in the customer's operation centers to improve ways in which the information is used and to effectively develop new methods for handling information as a business grows. (See Figure 1.)

FM—Today it's EDP takeover

The real world of FM is quite different from the ideal version just described, and there will be a period of long and difficult transition to reach that level. Actual takeover of an existing EDP installation is now the prime determinant of whether an FM relationship exists.

When the FM vector takes over the EDP installation, it also takes over such EDP dpeartment tasks as (1) possession, maintenance, and operation of all EDP equipment and the payment of all rental fees or acquisition of equipment ownership, (2) hiring and training all EDP personnel, and (3) development of applications, performance of systems analysis, acquisition of new equipment and implementation of new applications.

Takeover may be partial

Many FM vendors are increasingly offering cafeteria-style services so that the customer can retain control over EDP activities that he can perform proficiently. In some cases where equipment is owned, the customer may retain title to the equipment. Salaries of EDP personnel may continue to be paid by the customer, but responsibility for management is assigned to the vendor.

Also included as partial FM are takeovers of less than the client's total EDP activities. EDP activity of only a single division or a major application may be taken over by an FM vendor. Merely taking one of many applications on a computer and performing this function on a service bureau or time-sharing basis, however, is not included as an FM contract.

HOW EDP USERS BENEFIT

FM benefits: A study in contrast

Some FM users benefit . . .

Southwestern Life, a $5 billion life insurance company in Dallas and a customer of EDS, typifies the satisfied FM user. Southwestern Life's vice president, A. E. Wood, has stated, "We are very pleased with our agreement and the further we get into it, the more sure we are we did the right thing. We won't save an appreciable amount of money on operations, but the efficiency of operation will be improved in great measure. To do the same job internally would have taken us two to three times as long and we still would not have benefited from the continual upgrading we expect to see with EDS."

. . .and some do uot

Disgruntled users exist too, but they are more difficult to find and in many cases are legally restricted from discussing their experiences. One manufacturing company told us, "We cannot talk to you; however, let me say that our experience was unfortunate, very unfortunate. They (the FM vendor) did not understand our business, did not understand the urgency of turnaround time on orders. We lost control of our orders and finished goods inventory for six weeks. As a result we lost many customers whom we are still trying to woo back after more than a year."

Two medium-sized banks had similar comments that indirectly revealed much about FM benefits. "We're not in any great difficulty. In fact, the EDP operations now are running well, but every time we want to make a change it costs us. I wish I had my own EDP manager back to give orders to."

FM benefits are far ranging

Large users benefit least from FM

There is no question in our mind that there are many potential benefits for FM users. However, installation size is the primary yardstick for measuring benefits users can obtain from FM; large users have the least to gain for several reasons. In most cases, large users have already achieved economy-of-scale benefits which FM and other computer services can bring to bear. Large users typically have computers operating more shifts during the day and do not allow the computers to sit idle.

In addition to higher utilization, larger users can more fully exploit the capabilities of applications and systems programmers because they can spread these skills over more CPU's than can smaller users.

For these and other reasons, it is much more difficult to demonstrate to large users that an FM vendor can operate his EDP department more efficiently and less expensively. For these reasons, the bulk of FM revenue will come from the small- or medium-sized EDP user. This is defined as a user who has a 360/50 or smaller computer and is spending less than $1.5 million per year on EDP.

Improved EDP operations

The most tangible benefit FM can bring to an EDP user is improved control over the EDP operations and stabilization of the related operating costs. This conclusion is based on Quantum's field research which has shown that installations in the small-to medium-sized range are out of control despite the refusal by managers to admit it.

Lack of EDP planning, budgeting, and scheduling shows up in obvious ways, such as skyrocketing costs, as well as in obscure ways that are difficult to detect, yet contribute significantly to higher EDP costs. These subtle inefficiencies include program reruns due to operator or programming errors, equipment downtime due to sloppy programming documentation, and idle time due to poorly scheduled EDP workloads.

Because they are obscure and often hidden by EDP departments, it is difficult for managements in small and medium installations to detect and correct these problems. On the other hand, an FM vendor can often quickly identify these problems and offer corrective remedies because his personnel are trained to uncover these inefficiencies and his profits depend on their correction.

Smaller investments to upgrade EDP

FM can also benefit end users by reducing proposed future increases in EDP costs. Small- and medium-sized users that have a single computer must eventually face the problem of increasing their equipment capacity to meet requirements of revenue growth and expanded applications. This often means a significant increment in rental and other support costs. A 360/30 user, for example, who is spending $13,000 a year on equipment may have to jump to a 360/40 or a 370/135, costing $18,000–$22,000 per year to achieve the required increase in computing power.

	360/40	360/50	360/65
HAVE OS NOW	10%	46%	69%
HAVE NO OS NOW, BUT PLAN TO INSTALL IN 1971–72	48	38	8
HAVE NO OS NOW AND NO PLANS	42	16	23
TOTAL	100%	100%	100%

TABLE A—User OS Plans 1971-72

Support costs will also increase, in many cases more quickly. If a user is acquiring a 360/40, for example, he probably will have to use an Operating System (OS) to achieve efficient machine performance. Many users today will upgrade their software as shown in Table A. An OS installation requires a higher level of programming talent than is currently required to run a DOS 360 system. Because the user does not need the full time services of these system programmers, FM offers an economical solution whereby system programmers are shared among multiple users.

Elimination of EDP personnel problems

One of the most serious problems users encounter in managing EDP operations is personnel management. The computer has acquired an aura of mysticism that has tended to insulate the EDP department from the normal corporate rules and procedures. Many programmers often expect to receive special treatment, maintain different dress and appearance and obtain higher pay. High turnover among EDP personnel, often two to three times the norm for other company operations, further aggravates EDP personnel problems.

Through subcontracting, FM vendors can separate EDP personnel from the corporation and thus alleviate this situation for management.

Eased conversion to current generation software

Over one-third of all users are locked into using third generation computers in the emulation mode,

	ANNUAL EDP EXPENDITURES	PERCENT OF EDP COSTS SPENT ON PLANNING, ETC.
LARGE COMPANIES	>$1.5 MILLION	1–5%
MEDIUM COMPANIES	$300K–1.5 MILLION	0–2%
SMALL COMPANIES	< $300K	0–1%

TABLE B—User Expenditures on EDP Planning

where second generation language programs are run on third generation computers.

Although software conversion is a difficult and expensive task for users, the FM vendor who has an industry-oriented approach usually has a standard package already available that the customer can use. In several installations, FM vendors have simplified conversion, thus providing their users with the economies of third generation computers.

Improved selection of new equipment and services

Users of all sizes continually need to evaluate new equipment and new service offerings, including the evaluation of whether to buy outside or do in-house development.

Again, the large user holds an advantage because his size permits him to invest in a technical staff dedicated to evaluations. Installations spending more than $1.5 million annually for EDP usually have one full time person or more appointed to these functions. In smaller installations there is no dedicated staff and pro-tem evaluation committees are formed when required. Table B shows the relationship between the size of EDP expenditures and the share of those expenditures allocated to planning, auditing and technical evaluations.

In this area of EDP planning, FM can benefit users in two ways. First, FM vendors can and do take over this responsibility and, second, the effective cost to any single user is less because it is spread over multiple users.

Other operating benefits

One potential benefit from FM relates to new application development. Typically, 60 percent or more of a firm's EDP expenditures are tied to administrative applications, such as payroll, general accounting, and accounts payable. Because of the relatively high saturation in the administrative area, firms are now extending the use of EDP into operational areas such as production control and distribution management. However, many of these firms lack the qualified EDP professionals and line managers necessary to develop and implement applications in non-administrative areas. Thus, they have become receptive to considering alternatives, including FM.

Major EDP cost savings

Earlier in this chapter, the stabilization of EDP costs was discussed. Now we will focus on the major savings that FM can provide through the actual reduction of EDP costs. This potential FM benefit is too often the major theme of an FM vendor's sales pitch. Consequently, its emotional appeal often clouds a rational evaluation that should precede an FM contract.

If the FM contract is well written and does not restrict either party, the FM vendor *can* apply his economies of scale and capabilities for improving EDP operations and should be able to show a direct cost savings for the customer. However, these "savings" may be needed to offset costs of software conversion or other contingencies and thus, may not really be available to the customer in the early contract years.

Long range benefits—Better information

Improved operation control and profits through better information—this is the major long-range benefit from FM. While this contribution is not unique to FM vendors, few EDP users today have been able to develop a close relationship between company operations and EDP. Companies such as Weyerhauser and American Airlines—generally recognized as leading edge users—are few in number, and many try to emulate their achievements in integrating EDP into the company operations.

EDP expenditures, however, are seldom judged on their contribution toward solving basic company problems and increasing revenues and profits. Many apparently well-run EDP departments would find it difficult to justify their existence in these terms. The situation is changing, however.

An indication of this new attitude is the increased status of the top EDP executive in large firms. The top EDP executive is now a corporate officer in over 300 of the Fortune 500 firms. While titles often mean little, the change to Vice President or Director of Business Information from Director of EDP Operations suggests that top management in many companies has considered and faced the problem.

In addition to new management titles, continuing penetration of EDP functions into operating areas is increasingly evident.

FM—A permanent answer for users

FM should not be treated as an interim first-aid treatment for EDP. There are several good reasons for continuing the FM relationship indefinitely.

- Individual users cannot duplicate the economies of scale that FM vendors can achieve. Standard softwar epackages, for example, require constant updating and support and new equipment evaluations are constantly required if lowest cost EDP is to be maintained.
- Top management would have to become involved in EDP management if operations were brought back in-house. This involvement would take time from selling and other revenue producing activities. A rational top management trys to minimize the share of its time spent on cost-management activities.
- By disengaging from the FM contract, the customer risks losing control over his EDP again while receiving no obvious compensation for this risk. Even if the customer believes he is being overcharged by the FM vendor, there is no real guarantee the excess profits can be converted to savings to the customer.

For these reasons an FM relationship should normally be considered permanent rather than temporary.

MARKET STRUCTURE AND FORECAST

Current FM market

Total FM market size and recent growth

The 1971 market for FM services totals $645 million with 337 contracts. However, 45 percent or $291 million was captive and not available to independents. Captive FM contracts are defined as being solidly in the possession of the vendor because of other than competitive considerations. Typically, captive contracts are negotiated between a large firm and its EDP spinoff subsidiary.

The remaining market is available to all competitors and totals $354 or 55 percent. Available does not necessarily mean the contract is available for competition immediately, since most contracts are signed for a term of two to five years. Captive and available 1971 FM revenues and contracts are shown in Figure 2.

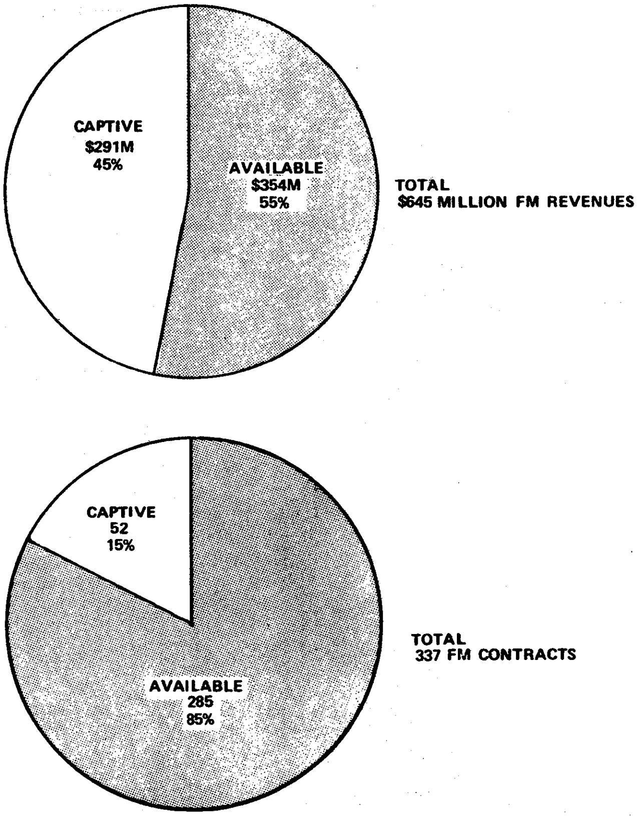


Figure 2—1971 FM market

Industry analysis

Discrete and Process Manufacturing are the largest industrial sectors using FM services and account for over 44 percent of total FM revenues. However, most EDP spinoffs have occurred in manufacturing and much of these FM revenues are therefore captive and not available to independent competitors. After deleting the captive portion, the two manufacturing sectors account for only 12 percent of the available 1971 FM market of $354 million.

Manufacturing has failed to develop into a major available FM market primarily because there is a general absence of common business and accounting procedures from company to company, thus, providing no basis for leveraging standard software. This is true even within manufacturing subsectors producing very common products. In the medical insurance sector, however, Federal Medicare regulations enforce a common method for reporting claims and related insurance data, thus providing a good basis for leveraging standard software.

The Medical Sector accounts for 25 percent of available FM revenues. The Medical Sector includes medical health insurance companies (Blue Cross/Blue Shield)

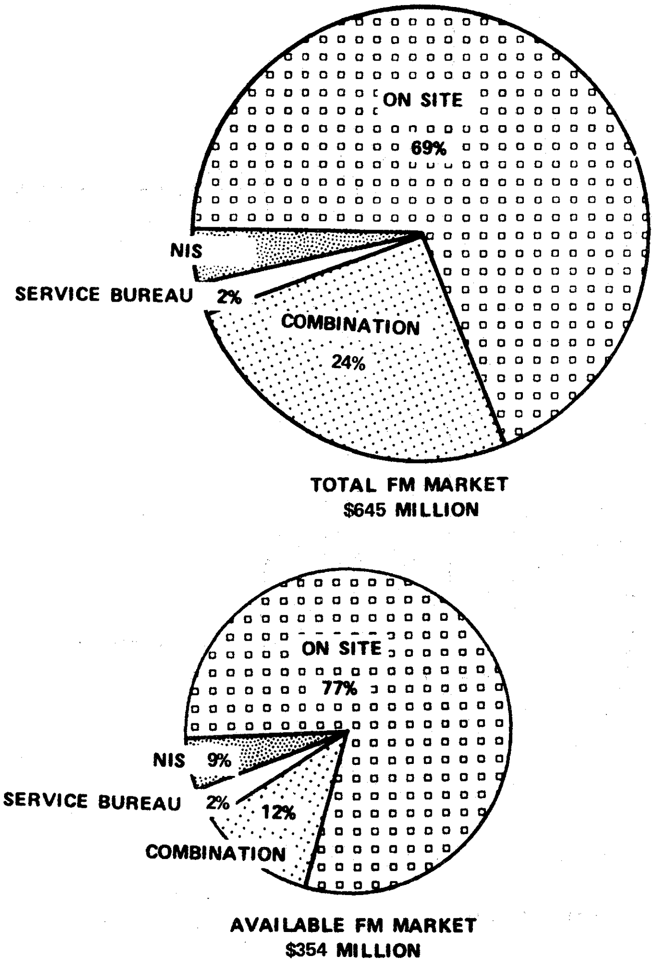


Figure 3—1971 FM market by type of performance

and hospitals. This sector was the first major commercial FM market. FM continues to be attractive in this sector because it permits rapid upgrading of EDP to meet the new Medicare reporting procedures and relieves the problem of low EDP salary scales.

The largest industry sector in the available FM market, the Federal Government, accounts for over 34 percent of available revenues. All Federal Government contracts are awarded on the basis of competitive bids. Most Federal Government FM contracts still tend to be purely subcontracting of EDP operations rather than total business information management which is becoming more common in commercial markets.

The Finance Sector currently accounts for 22 percent of available FM revenues. Banks and insurance companies are the major markets within the Finance Sector which also includes brokerage firms, finance companies and credit unions.

Type of performance

FM vendors who initially take over on-site operation of a customer's computer strive for economies of scale. This has created a trend whereby the FM vendor has eliminated the need for the customer's computer by processing data through NIS (timesharing) or service bureaus.

NIS now accounts for 5 percent of total FM revenues. Service bureau processing which requires the physical transport of data from the client's location to the vendor's computer installation accounts for 2 percent of total FM revenues. In Figure 3, which depicts FM market by type of performance, combination refers to the use of two or more of the above services to carry out the FM contract.

Types of vendors

Types of vendors that perform FM contracts are described below:

- *Independents* who accounted for 67 percent of total FM revenues in 1971 were startups in the computer service industry or vendors who have graduated from the ranks of spinoffs.
- *Spinoffs* are potentially strongest in their "home" industries; however, competitive pressures may limit market penetration here. An oil company spinoff, for example, would have a difficult time selling its seismic services to another oil company because of the high value placed on oil exploration and related information.
- *Computer manufacturers* are increasingly offering FM services. Honeywell has several FM contracts and will be joined by Univac and CDC who have announced intentions to market FM services. The RCA Services Division should find FM opportunities among RCA customers. IBM has several ways in which it can enter FM, and will show an expanding profile.

Contract Values

The average FM contract in 1971 is valued at slightly less than $2 million. This is the equivalent of a user with two or three computers, one at least a 360/50. However, this is based on the total market analysis which distorts the averages for captive and available FM markets. An analysis of available and captive contracts shows that the average value of an available contract drops to $1.24 million, which would be equivalent to a user with

TABLE C—Major Vendors

	TOTAL		AVAILABLE	
Rank	Company	FM Revenues*	Company	FM Revenues*
1	Electronic Data Systems Corp.	95.7	Electronic Data Systems Corp.	95.7
2	McDonnell Douglas Automation Co.	89.4	Computer Sciences Corp.	26.6
3	Boeing Computer Services Inc.	82.2	Boeing Computer Services Inc.	22.2
4	University Computing Company	42.2	Computing and Software, Inc.	14.7
5	Computer Sciences Corp.	26.6	System Development Corp.	12.8
6	Grumman Data Systems	25.6	University Computing Company	10.1
7	Computing and Software, Inc.	14.7	National Sharedata Corp.	7.5
8	System Development Corp.	14.0	McDonnell Douglas Automation Co.	7.1
9	Martin Marietta Data Systems	11.5	Executive Computer Systems, Inc.	5.2
10	Westinghouse Tele-Computer Systems Corp.	11.0	Cambridge Computer Corp.	4.4
11	MISCO	10.0	Greyhound Computer Corp.	4.3
12	National Sharedata Corp.	7.5	Tracor Computing Corp.	4.0
13	A. O. Smith Corp.'s Data Systems Div.	5.4	Programming Methods, Inc, (PMI)	3.8
14	Executive Computer Systems, Inc.	5.2	MISCO	3.0
15	Unionamerica Computer Corp.	5.0	Allen Babock Computing Corp.	2.9
16	Cambridge Computer Corp.	4.4	RAAM Information Services Corp.	2.5
17	Greyhound Computer Corp.	4.3	Data Facilities Management Inc.	2.3
18	Tracor Computing Corp.	4.0	Bradford Computer and Systems, Inc.	2.0
19	Mentor Corp.	4.0	Computer Usage Co., Inc.	2.0
20	Programming Methods, Inc. (PMI)	3.8	Martin Marietta Data Systems	1.5

* Annual Rate in 1971 in millions of dollars

two 360/40's. Analysis of captive contracts, however, shows that the average value is significantly higher at $5.6 million per year. Most of these contracts are spinoffs from large industrial firms who have centralized computer installations or multiple installations spread throughout the country.

A more revealing analysis of contract values is shown in Table D. Here total and available contracts are distributed according to contract value. From this analysis, it is clear that well over one-third of contracts are valued at $300,000 or less per year. A typical computer installation of this value would include a 360/30, 360/20, 360/25 or equivalent computers in other manufacturers' lines. There are in total 18 contracts, captive or available, valued at more than $5 million per year. These are all spinoff parent or Federal Government contracts.

CONTRACT VALUE PER YEAR	ALL CONTRACTS #	%	AVAILABLE CONTRACTS #	%
0.1–0.3	129	38.3	122	42.8
0.31–0.5	40	11.9	30	10.5
0.51–0.8	39	11.6	34	11.9
> 0.8	129	38.2	99	34.8
TOTAL	337	100.0	285	100.0

AVERAGE CONTRACT VALUE: $1.91 MILLION
285 AVAILABLE CONTRACTS – AVERAGE VALUE: $1.24 MILLION
52 CAPTIVE CONTRACTS – AVERAGE VALUE: $5.61 MILLION

TABLE D—FM Contract Analysis by Value

Projected 1977 FM market

FM market potential

The ultimate U.S. market potential for FM is the sum of EDP expenditures for all users. By 1977 EDP expenditures for equipment, salaries and services will total $29.5 billion spread among 52.4 thousand users. Since FM benefits are not available to all users, five criteria were developed to help identify the industry sectors which could most benefit from FM and would be most amenable to accepting FM as an alternative approach for EDP. The five criteria are:

- *Homogeneous Business Methods.* Industries with similar information requirements from company to company are ideal situations for FM. These might be the coding of business records, such as the MICR codes used in banks, or price standards, e.g., tariffs used in motor freight.

TABLE E—High Growth Potential FM Markets

Selection Criteria Industry Sector	Homogeneous Business Methods	Similar Products or Services	Regulation by Government Agencies	Prior Evidence of Subcontracting Services	Special EDP Operating Problems
Medical—Health Insurance	X	X	X	X	X
Federal Government		X	X	X	X
Banking—Commercial	X	X	X	X	X
Insurance	X	X	X		X
State and Local Government		X	X	X	X
Motor Freight	X	X	X		X
Brokerage	X	X	X		X
Utilities—all except telephone	X	X	X		X
Medical—Hospitals, Clinics		X	X	X	X
Regional/Interstate Airlines	X	X	X		X
Mutual Fund Accounting	X	X	X		X
Banking—Savings	X	X	X		
Small & Medium Aerospace Cos.	X		X	X	
Education—Elementary and Secondary		X		X	
Education—College		X		X	
Construction and Mining		X	X		

- *Similar Products or Services.* The more similar the products and services sold by companies within a given industry sector, the more likely they will have common business procedures and, therefore, EDP systems. In the brokerage industry, for example, there is little differentiation in the serivce provided.
- *Regulation by Government Agencies.* Industries that are regulated directly by State/Federal agencies or indirectly through strong trade associations also become good candidates for FM because of the enforced standards for pricing, operating procedures, account books, or other factors that impact EDP operations. Health insurance firms, utilities of all types, and brokerage firms are typical of these highly regulated industries.
- *Prior Evidence of Subcontracting Services.* Prior company or industry practices which indicate that subcontracting of vital services is an accepted business procedure also help pinpoint industries with high FM potential. Correspondent relationships between smaller banks and larger city banks, historically a part of the banking industry, is an example.
- *Special EDP Operating Problems.* Several industries have special EDP operating problems. These may result from historically low pay scales for EDP personnel which cannot easily be changed, such as in state and local government or a pending major conversion in basic accounting approaches imposed by an outside force, resulting in major EDP conversions as was the case in health insurance when Medicare and state health programs were implemented in the 1960's.

TABLE F—Total FM Revenue Growth 1971–1977

	1971		1977		Compound Annual Growth Rate of FM Revenues
	Revenues $ Millions	Contracts #	Revenues $ Millions	Contracts #	
Medical and Other	104	89	446	635	27
Finance	88	89	590	590	37
Discrete Manufacturing	146	42	389	255	18
Process Manufacturing	144	34	344	200	16
Government—State and Local	7	15	350	350	92
Government—Federal	122	36	236	185	12
Utilities and Transportation	20	13	318	400	58
Wholesale and Retail Trade	13	17	160	320	52
EDP Service Bureaus and NIS Operators	1	2	23	60	69
Total	645	337	2856	2995	28

TABLE G—Available FM Revenue Growth 1971-1977

	1971		1977		Compound
	Revenues	Contracts	Revenues	Contracts	Annual Growth Rate of FM
	$ Millions	#	$ Millions	#	Revenues
Government—State and Local	7	15	350	350	92
Finance	77	76	503	480–510	37
Medical and Other	88	81	280	380–400	21
Government—Federal	122	36	236	185	12
Wholesale and Retail Trade	12	16	133	250–270	49
Utilities and Transportation	3	6	196	235–245	137
Discrete Manufacturing	17	32	135	90–95	41
Process Manufacturing	27	22	129	70–75	30
EDP Service Bureaus and NIS Operators	1	1	6	12–14	35
Total	354	285	1968	2052–2144	33

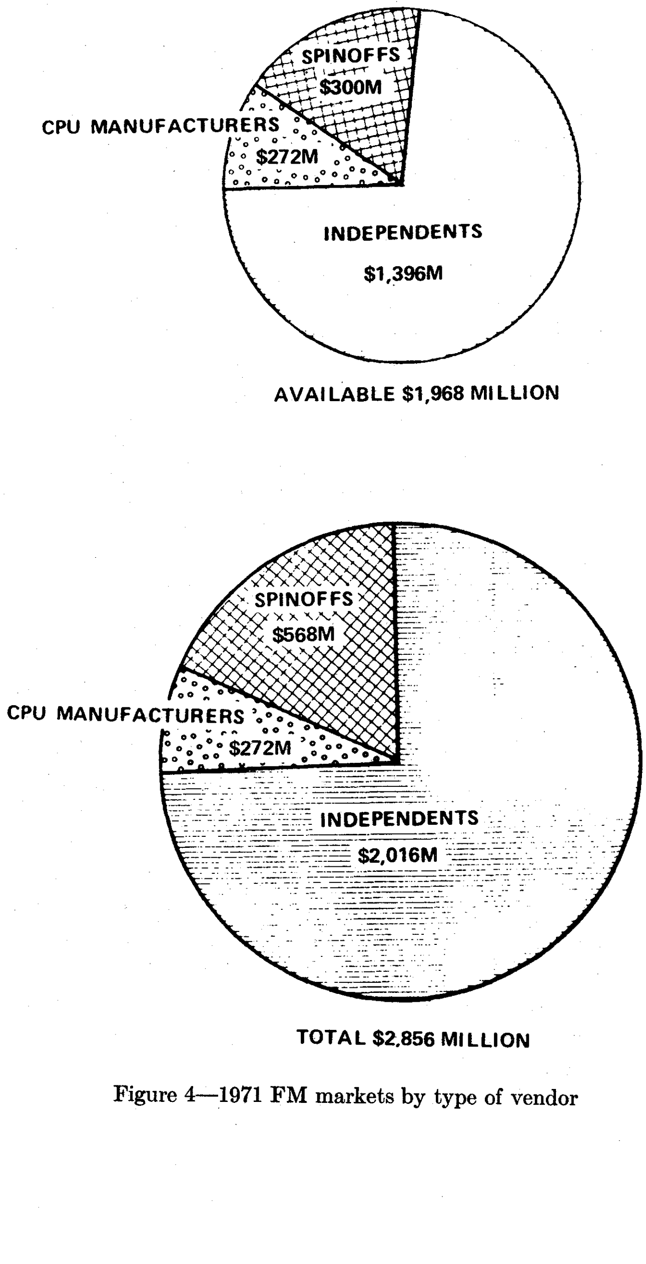


Figure 4—1971 FM markets by type of vendor

The above criteria were applied against major industry sectors. As a result, 16 sectors were identified and ranked according to their suitability for FM. (See Table E.)

On the basis of this analysis the industry sectors most likely to benefit from FM include banking (mainly commercial), insurance, state and local governments, Federal Government, motor freight, brokerage, and medical (hospitals, and health insurance firms). Of these, the Federal Government and medical sectors are already established FM markets and will grow more slowly as a result.

Projected FM revenues, 1971-1977

Actual realized FM revenues will be $2.86 billion in 1977. This is a 28 percent annual growth from $645 billion in 1971. Total contracts will increase to 2,995 in 1977 from 337 in 1971, with an average contract value of slightly less than $1 million.

The available portion of the 1977 FM market will total $1.97 billion, up from $354 million in 1971, a growth of over 500 percent. Related contracts will be between 2,000 and 2,200 in 1977, up from 285 in 1971. See Tables F and G.

Who are the vendors?

Independent vendors will retain the same share of the total FM market in 1977, as in 1971. Computer manufacturers will increase their penetration in the FM business primarily to protect installations that are threatened by competitive equipment. See Figure 4.

HOW TO EVALUATE FM PROPOSALS

Know what benefits are desired

For the purposes of reading this segment, assume you are an EDP user considering a proposed FM contract. Assume further that by reading the previous material, you have concluded that, indeed, FM can benefit your company, both in terms of improved EDP operations and in improved information flow to the operating departments.

But now you must get specific about the vendor, his proposal, and finally the detailed provisions of the contract he wishes you to sign. In this chapter we will provide the guidelines you can use to make these evaluations.

Before digging into the evaluation guidelines, you should first articulate just what you, the management, and the current EDP department are expecting in the way of benefits. By doing this, you can compare your expectations as a customer with what the FM vendor is willing and able to provide.

Have you had a poor experience with EDP? Is your primary objective to get out of the operating problems of an EDP department? If this is the case, then don't expect immediate improvements in the information you are receiving from EDP and the speed in which it flows to your operating departments—even if you have been told by the FM vendor this is to be the case. On the other hand, if your real goal is speeding order entry and decreasing finished goods inventory by a factor of three without a major investment in new applications software, then these are the points an FM vendor should be addressing in his proposal and you will want to evaluate him on this basis.

Assuming you and the vendor have agreed on a set of expectations, let us look at the guidelines you can use in evaluating the vendor, his proposal and the FM contract.

Evaluating vendor and his proposal

Vendor

Three potential problem areas should be explored to accurately appraise an FM vendor. These are: financial stability, past FM record, and level of industry expertise.

- Financial Stability

  Financial stability of the vendor is a critical issue to pin down, for if he is in difficulty, such as being short of working capital, your information flow from EDP could be stopped leaving you in an extremely serious and vulnerable position.
- Previous FM Performance Record

  Next to the financial record, the vendor's previous performance in FM as well as in other data services can be a good guide to his future performance on your contract. If the vendor has done well in past contracts, he no doubt will use his past work as a "showcase" and invite your visit to current sites he has under contract. However, the absence of these referrals should not be taken negatively due to the possible proprietary nature of current FM work.
- Industry Expertise

  Full knowledge of your industry and its detailed operating problems should be demonstrated completely by the vendor. This should include full appreciation for the operating parameters most sensitive for profitability in your industry and company. The vendor should be staffed with personnel who have had top management experience in the specific industry and people who have had experience in other specific industries. Vendors become more credible if they can show existing customers who are pleased with the vendor's services and who will testify to his ability to solve specific industry-oriented EDP problems.
- Proposal Responsiveness

  The proposal should be addressed specifically to the objectives that you and the vendor agreed were the purpose of considering the FM contract. The vendor should detail exactly how he will improve your EDP operation or provide faster or improved information to serve your operating areas. He should suggest where savings can be made or what specific actions he can take that are not now being taken to effect these savings.
- Work Schedule for Information Reports

  While it is not desirable to pin the vendor down to an operating schedule for the EDP department—for it is exactly this flexibility that allows him to achieve economies of scale—he should, however, be very specific about the schedule for delivery of required reports. If you have a data-entry problem, for example, then the proposal should indicate that the computer will be available when you need to enter data. The work schedule should fully reflect as closely as possible the current way in which you do business and any change should be fully justified in terms of how it can improve the operation of the whole company, not just the operation of the EDP department.
- Equipment Transfer

  Details of equipment ownership and any transfers to the vendor should be specified. Responsibility for rental or lease payments should also be detailed. Responsibility for maintenance not built into equipment rentals or leases should also be delineated.

- Cost Schedule

  Contract pricing is the most critical cost item. A fixed-fee contract is advantageous to both parties if the customer's business volume is expected to continue at current levels or grow. If business drops, a fixed fee could hurt the customer. Thus, the fairest pricing formula is composed of two components: a fixed fee to cover basic operating costs and a variable fee based on revenue, number of orders, or some easily identifiable variable sensitive to business volume. Some contracts also include a cost-of-living escalator.

  The proposed cost schedule should also take into account equipment payments, wages, salary schedules, travel expenses, overhead to be paid to the customer (if the vendor occupies space in the customer's facilities) and all other expenses that might occur during the course of the contract. If special software programming or documentation is to be performed for the customer, the hourly rates to be charged should be identified in the contract.

- Vendor Liaison

  A good proposal recognizes the need for continuing contact between top management and the FM vendor. Close liaison is especially required in the early days of the contract, but also throughout its life. The cost for this liaison person should be borne by the customer, but the responsibilities and the functions that will be expected of him should be clearly stated in the proposal.

- Personnel Transfer

  Since all or most of the personnel in EDP operations will be transferred to the FM vendor, you must make sure that this will be an orderly transfer. Several questions arise in almost every contract situation and should be covered in the proposal: Does the proposal anticipate the possible personnel management problems that might come about? If all personnel are not being transferred and some may be terminated, how will this be handled? Are FM vendor personnel policies consistent with yours? Has the vendor taken into account the possibility that large numbers of persons may not wish to join the vendor and may leave?

- Failure to Perform

  While it is most desirable to emphasize the positive aspects of an FM relationship, the negative possibilities should be explored to the satisfaction of both parties. Most of these revolve around failure to perform. If the vendor fails to perform his part of the contract, you should be able to terminate the contract. The proposal should detail how this termination can be carried out. Is the vendor, for example, obligated to permit you to recover your original status and reinstall your in-house computer? What are the penalties the vendor will incur? What is the extent of his liabilities to replace lost revenue, lost profits that you may suffer as a result of his failure to perform? How will these lost revenues and lost profits be identified and measured?

  That's the vendor's side, but you also have obligations as a customer. If your input data is not made available according to schedule, for example, what is your possible exposure in terms of late reports?

- Software

  It is important to pin down ownership of existing software when an FM contract is signed and any subsequent software that is developed. Proprietary as well as non-proprietary software packages should be identified and specified in the report so that competitors may not benefit unfairly if the FM vendor uses the packages with other clients in your industry.

  Software backup and full documentation procedures should also be identified. This is one area in which FM may be a great help. If your installation is typical, your backup and documentation procedures are weak and an FM vendor, using professional approaches, should be able to improve your disaster recovery potential.

FM contract: Marriage of porcupines

The FM contract should incorporate all the above issues, plus any others which are uniquely critical, in an organized format for signing. The FM contract is as legal a document as any other the company might enter into; therefore, the customer's legal staff should carefully review it in advance of any signing.

The body of a typical FM contract shows the general issues which have been discussed above and which apply in most FM contract situations. Attachments are used to detail specific information about the customer that is proprietary in an FM contract Attachments discuss the service and time schedule, equipment ownership and responsibility, cost schedule and any special issues.

One of the most striking features is the general absence of detailed legal jargon. This is typical in most

FM contracts and is a result of two factors. First, the two parties have attempted to communicate with each other in the language that both understand. Second, the wording reflects an aura of trust between the two parties. In a service subcontracting relationship the customer must implicitly trust the vendor. Without this mutual trust, it would be foolish for a vendor or a customer to even consider a proposal.

BIBLIOGRAPHY

EDP productivity at 50%?
Administrative Management June 1971 pp 67-67

EDP—What top management expects
Banking April 1972 pp 18-32

Facilities management users not sure they're using—If they are
Datamation January 1 1971 p 54

KUTTNER et al
Is facilities management solution to EDP problems?
The National Underwriter January 23 1971

H C LUCAS JR
The user data processing interface
Working Paper #177 Graduate School of Business Stanford University

P J McGOVERN
Interest in facilities management—Whatever it is—Blossoms
EDP Industry Report April 30 1971

D M PARNELL JR
A new concept: EDP facilities management
Administrative Management September 1970 pp 20-24

I POLISKI
Facilities management—Cure-all for DP headaches?
Business Automation March 1 1971 pp 27-34

A RICHMAN
Oklahoma bank opts for FM
Bank Systems and Equipment February 1970 pp 18-32

L W SMALL
Special report on bank automation
Banking April 1971

When EDP goes back to the experts
Business Week October 18 1969 pp 114-116

Quantum Science Corporation Reports
Dedicated information services July 1970
Facilities management—How much of a gamble? November 1971
Federal information services October 1971
Network information services April 1971

Automated map reading and analysis by computer

by R. H. COFER and J. T. TOU

University of Florida
Gainesville, Florida

INTRODUCTION

A great deal of attention is presently being given to the design of computer programs to recognize and describe two-dimensional pictorial symbology. This symbology may arise from natural sources such as scenery or from more conventionalized sources such as text or mathematical notation. The standardized graphics used in specification of topographic maps also form a conventionalized, two-dimensional class of symbology.

This paper will discuss the automated perception of the pictorial symbology to be found within topographic maps. Although conventionalized, this symbology is used in description of natural terrain, and therefore has many of the characteristics of more complex scenery such as is found within aerial photography. Thus it is anticipated that the techniques involved may be applied to a broader class of symbology with equal effectiveness.

The overall hardware system is illustrated by Figure 1. A map region is scanned optically and a digitized version of the region is fed into the memory of a computer. The computer perceives in this digitized data the pictorial symbology portrayed and produces a structured output description. This description may then be used as direct input to cartographic information retrieval, editing, land-use or analysis programs.

THE PROGRAM

Many results of an extensive research into the perception of pictorial symbology have been incorporated into a computer program which recognizes a variety of map symbology under normal conditions of overlap and breakage. The program is called MAPPS since it performs Machine Automated Perception of Pictorial Symbology. MAPPS is written in the PL/1 programming language heavily utilizing the language's list and recursive facilities. It is operated on the University of Florida's IBM 360/65 computer utilizing less than 100K words of direct storage.

Although the set of possible map symbols is quite large, those used in modern topographic maps form the three classes shown in Figure 2. Point symbology is used to represent those map features characterized by specific spatial point locations. This class of symbology is normally utilized to represent cultural artifacts such as buildings, markers, buoys, etc. Lineal symbology is used to mark those features possessing freedom of curvature. This class is normally utilized to represent divisional boundaries, or routes of transportation. Typical examples of lineal symbology include roads, railways, and terrain contours as well as various boundary lines. Area symbology is used to represent those features possessing homogeneity over some spatial region. It is normally composed of repeated instances of point symbology or uniform shading of the region. Examples include swamps, orchards, and rivers.

As its extension to the recognition of area symbology is rather straightforward, MAPPS has been designed to recognize the point and lineal forms of symbology only. Further it has been designed to recognize only that subset of point and lineal symbology which possess topographically fixed line structures. This restriction is of a minor nature since essentiall all map symbology is, or may be easily converted to be, of this form. Even given these restrictions, MAPPS has immediate practical utility since many applications of map reading require only partial recognition of the symbology of a given map. As an example, the survey of cultural artifacts can be largely limited to the recognition of quite restricted forms of point and lineal symbology.

Color information provides strong perceptual clues in maps. On standard topographic maps for instance blue represents hydrographic features, brown represents terrain features, and black represents cultural features. Even so, utilization of color clues is not incorporated into MAPPS. This has been done to provide a more stringent test of other more fundamental techniques of

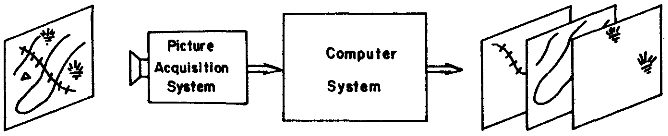


Figure 1—The overall hardware system

recognition. It is obvious however, that utilization of color descriptors can be easily incorporated, and will result in increased speed of execution and improved accuracy of recognition.

MAPPS is divided into three systems: picture acquisition, line extraction, and perception. In brief, the picture acquisition system inputs regions of the map into the computer, the line extraction system constructs data entities for each elementary line and node present in the input, and the perception system conducts the recognition of selected symbology. A flow-chart of MAPPS is shown in Figure 3.

PICTURE ACQUISITION

The picture acquisition system PIDAC is a hardware system developed by the authors to perform precision scanning of 35 mm transparencies within a research environment.[1] It consists of a flying-spot scanner, minicomputer, disk memory, storage display, and incremental tape unit. In operation, PIDAC scans a transparency, measures the optical density of the transparency at each point, stores the results in digital form, performs limited preprocessing actions, and generates a digital magnetic record of the acquired data for use by the IBM 360/65 computer.

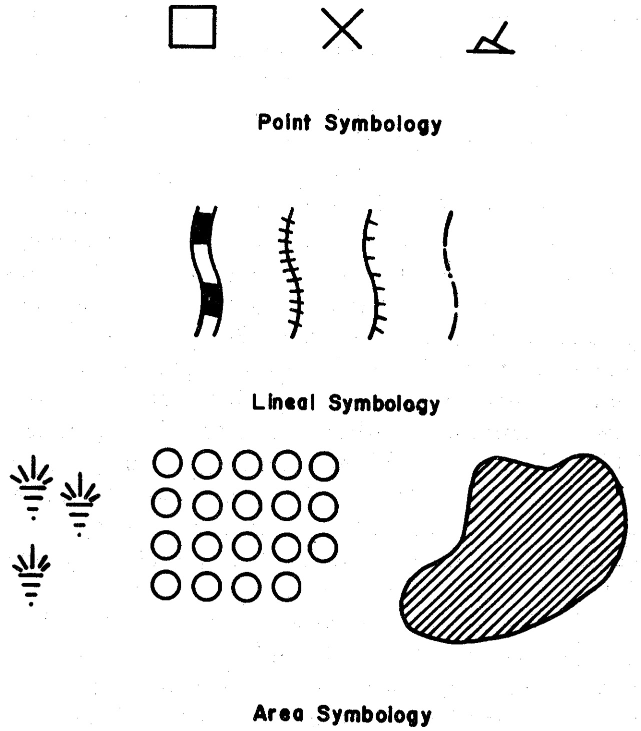


Figure 2—Classes of map symbology

For each transparency, PIDAC scans a raster of 800 rows by 1024 columns, a square inch in the film plane thus corresponds to approximately 10^6 points. At each raster point PIDAC constructs a 3-bit number corresponding to the optical density at that point. As the original map may be considered to be black and white, a preprocessing routine, operating locally, dynamically reduces the 3-bit code to a 1-bit code in a near optimal fashion. This action is accomplished by a routine called COMPACT since it compacts storage requirements as well. The result is an array whose elements correspond to the digitized points of the map region. This compacted array is then input to the line extraction system.

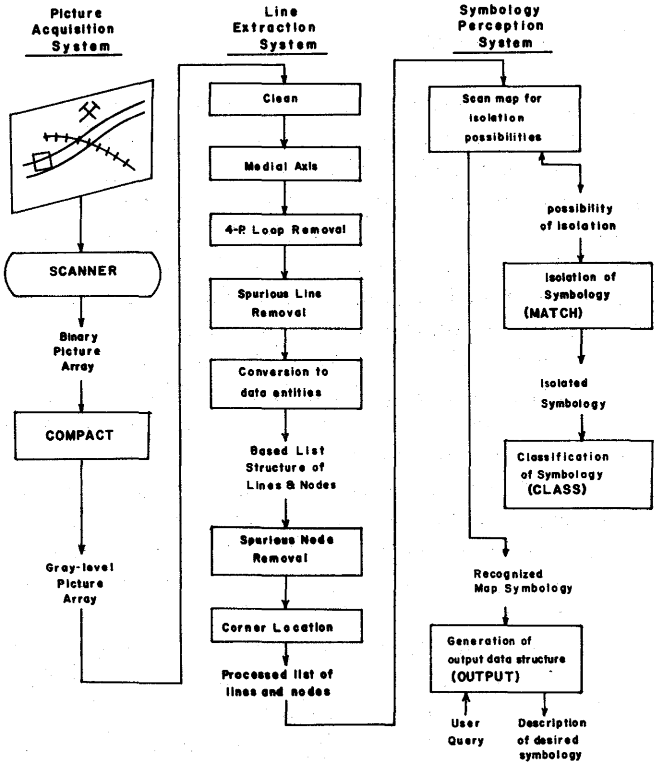
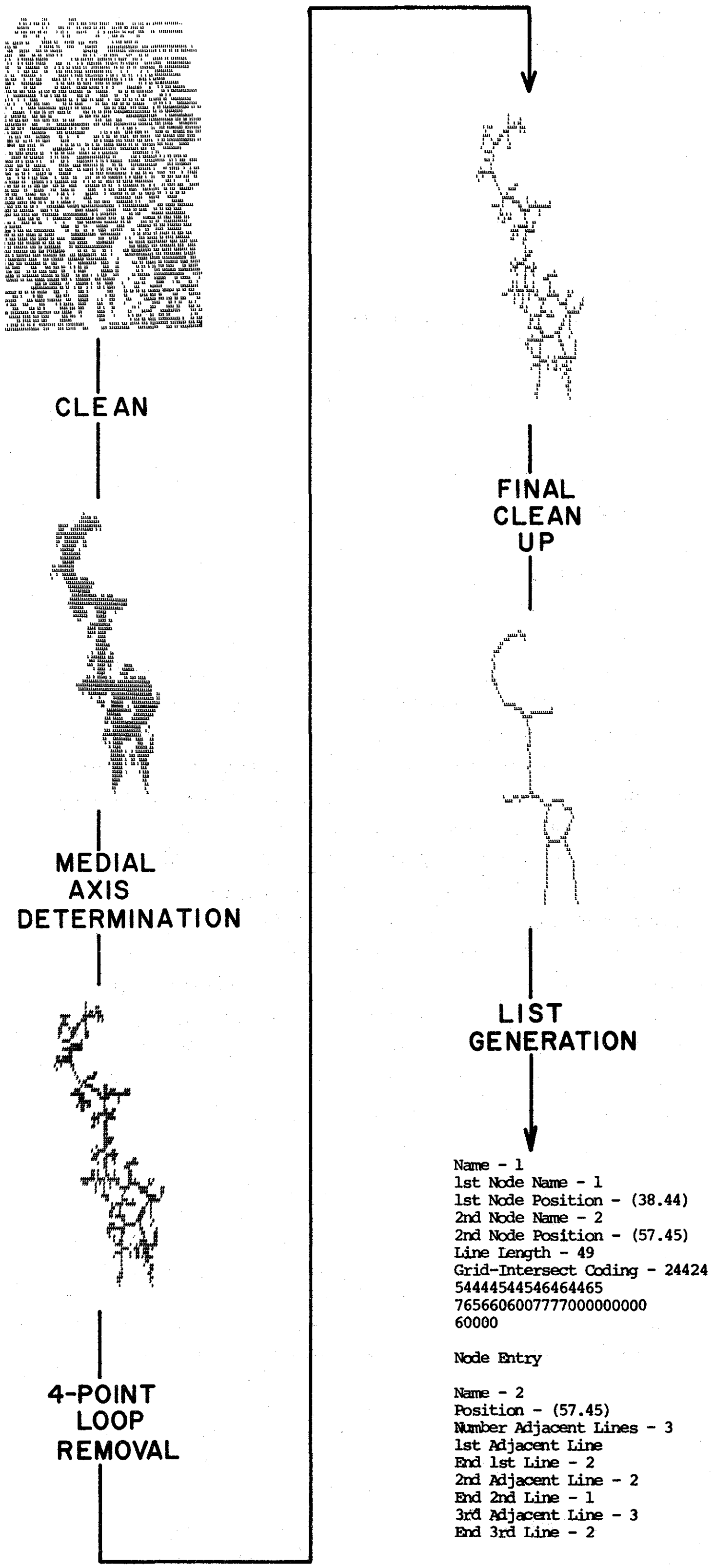

Picture Acquisition System
Line Extraction System
Symbology Perception System
SCANNER
Binary Picture Array
COMPACT
Gray-level Picture Array
Clean
Medial Axis
4-R Loop Removal
Spurious Line Removal
Conversion to data entities
Based List Structure of Lines & Nodes
Spurious Node Removal
Corner Location
Processed list of lines and nodes
Scan map for isolation possibilities
possibility of isolation
Isolation of Symbology (MATCH)
Isolated Symbology
Classification of Symbology (CLASS)
Recognized Map Symbology
Generation of output data structure (OUTPUT)
User Query
Description of desired symbology

Figure 3—MAPPS flow chart



```
Name - 1
1st Node Name - 1
1st Node Position - (38.44)
2nd Node Name - 2
2nd Node Position - (57.45)
Line Length - 49
Grid-Intersect Coding - 24424
544445445446464465
7656606007777000000000
60000

Node Entry

Name - 2
Position - (57.45)
Number Adjacent Lines - 3
1st Adjacent Line
End 1st Line - 2
2nd Adjacent Line - 2
End 2nd Line - 1
3rd Adjacent Line - 3
End 3rd Line - 2
```

Figure 4—Acti on of line extraction system

LINE EXTRACTION

As shown by Figure 3, the compacted array is input to the line extraction system. The function of this system is the extraction of each of the elementary line segments represented in the map, so that the program can conduct perception in terms of line segments and nodes rather than having to deal with the more numerous set of digitized points.

The system of line extraction, as developed, does not destroy or significantly distort the basic information contained within the map. This is necessary since significant degradation makes later perception more difficult or impossible. The action of the line extraction system is illustrated in Figure 4. First the map is cleared of all small holes and specks likely to have resulted from noise. Then a connected medial axis network of points is obtained for each black region of the map. This first approximation to a desired line network is converted to a true line network by an algorithm called 4-point loop removal. Operating on a more global basis, later algorithms remove spurious lines and nodes, locate possible corner nodes, and convert to a more suitable list processing form of data base. For each line and node, a PL/1 based structure is produced. Each structure contains attributes and pointers to adjacent and nearby data entities. The structure for a line entity contains the attributes of width, length, grid interest coding, as well as pointers to adjacent nodes. The structure for each node entity contains the attribute of position and pointers to adjacent lines and nearby nodes.

The line extraction system, being somewhat intricate, has been discussed in detail in a prior paper.[2] Abstractly, each state S of the system can be viewed as responding to distortions occurring within the map. These distortions may be characterized by a set of context sensitive productions of the form

$$R_l^m(i,j)R_l^n(i,j) \rightarrow R_l^m(i,j)R_l^{n'}(i,j) \qquad \iota=1,2,\ldots,N_S$$

$R_l^m(i,j)$ represent some region about the point (i,j) having a fixed size and gray-level distribution. $R_l^n(i,j)$ and $R_l^{n'}(i,j)$ represent regions of the point (i,j) having the same fixed size but differing gray-levels. By inversion of the production sets, each stage can be described as the repetitive application of the rules

$$R_l^m(i,j)R_l^{n'}(i,j) \rightarrow R_l^m(i,j)R_l^n(i,j) \qquad l=1,2,\ldots,N_S$$

in forward raster sequence to the points (i,j) of the map until no further response is obtainable. As an example, one such rule

$$\{M(i+1,j)=0,\ M(i-1,j)=0\}\ \{M(i,j)=1\}$$
$$\rightarrow \{M(i+1,j)=0,\ M(i-1,j)=0\}\ \{M(i,j)=2\}$$

is used in the medial axis determination to mark object regions of width 1 as possible line points for further investigation.

It is important to observe the degree of data reduction and organization which is accomplished through the extraction of line data. As previously mentioned, even a small map region contains a huge number of nearly 10^6 picture points. The extracted list structure typically contains no more than 300 lines and node points thereby resulting in a very significant data reduction. Equally significant, the data format of the list structure permits efficient access of all information to be required in the perception of symbology. The digitized map array therefore may be erased to free valuable storage for other purposes.

PERCEPTION OF SYMBOLOGY

It is interesting to observe that certain familiar pattern recognition procedures cannot be directly used in the recognition of map symbology. This results from the fact that in cartography, symbology cannot be well isolated as there are often requirements for overlap or overlay of symbology in order to meet spatial positioning constraints. Many of the techniques used for recognition of isolated symbology, such as correlation or template matching of characters, cannot be used to recognize such non-isolated symbology and are thus not very powerful in map reading. In MAPPS, alternative techniques have been employed to accomplish isolation of symbology in addition to its recognition.

THE CONCEPT OF ISOLATION PROCESSING

Conceptually, isolation of symbology from within a map cannot be accomplished in vacuo. Isolation requires some partial recognition, while recognition generally requires some partial isolation. This necessitates the use of a procedure in which isolation is accompanied by partial recognition. In order to guide this procedure, there must exist some *a priori* knowledge about the structure of the symbology being sought. The underlying structure of pictorial symbology, such as is present in maps and elsewhere, is found to be that of planar graphs upon the removal of all metric constraints. Using this structure the isolation process functions by sifting through the data of the map proposing possible instances of pattern symbology on a graph-theoretic equivalency basis; thereby suppressing extraneous background detail.

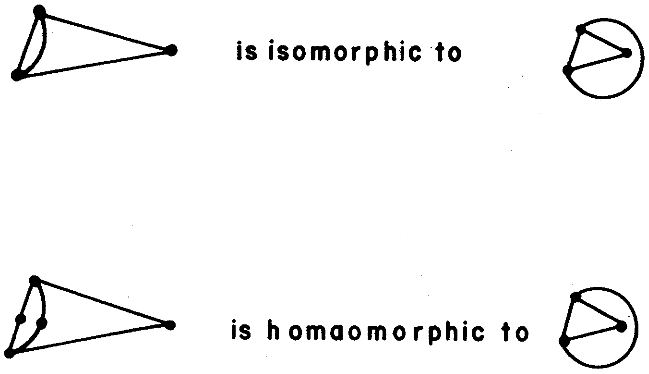


Figure 5—Graph equivalencies

Two types of graph equivalency are used in isolation. These are

- isomorphism
- homomorphism

One graph is isomorphic to another if there exists a one-to-one correspondence between their nodes which preserves adjacency of edges. A graph is homomorphic to another if it can be obtained from the other by subdivision of edges. Figure 5 yields an instance of isomorphic and of homomorphic equivalence of graphs.

Using the above definitions of graph equivalency, the process of isolation can be achieved by means dependent upon and able to cope with the types of structural degradation, Figure 6, found within actual maps. For instance, should a map contain no structural degradation, then on the basis of graph structures only, it is necessary and sufficient to propose as possible symbology isolations those components of the map which are isomorphic to the symbology being sought. If the map

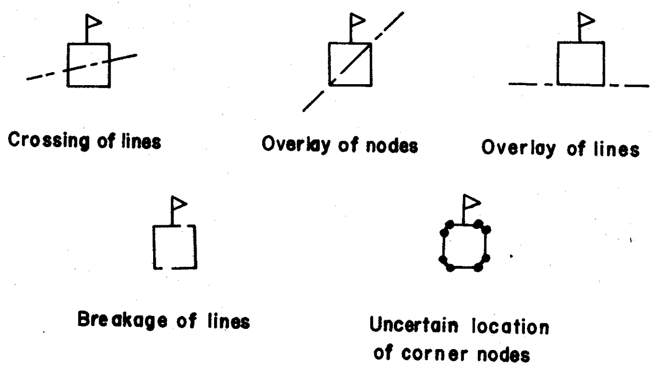
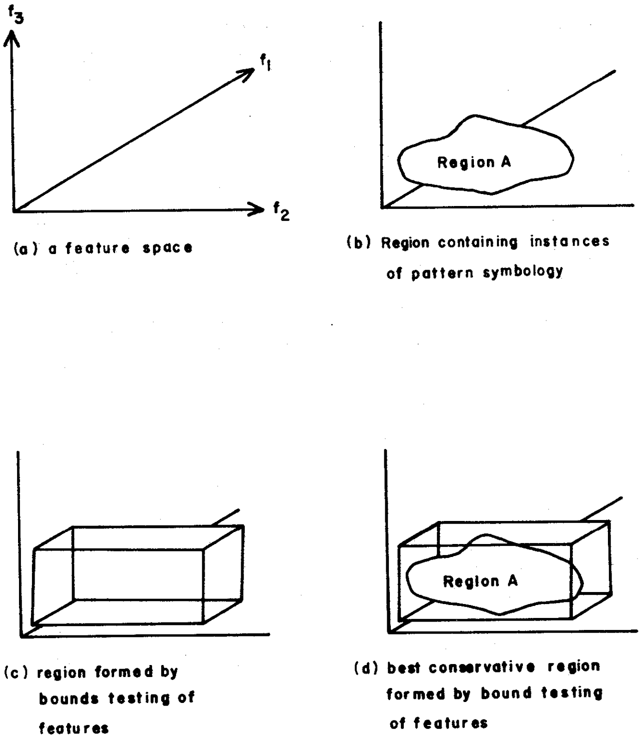


Figure 6—Structural degradations occurring in MAPPS

contains no crossing, overlay, or breakage of lines then on the basis of graph structures only, it is necessary and sufficient to propose as possible symbology isolations those partial subgraphs of the map which are isomorphic to the symbology. If the map contains no breakage or overlay of lines, then it is necessary and sufficient to propose those partial subgraphs which are homomorphic to the symbology sought. Finally, if a map contains as the only forms of structural degradation: line crossovers, line breakage, node overlay, and uncertain location of corner nodes, first complete the map by filling in all possible instances of line breakage. Then it is necessary and essentially sufficient to propose as possible pattern isolations those partial subgraphs of the completed map which have no two adjacent broken edges and which are homomorphic to the symbology sought.

Although the process of isomorphic matching of graphs can be conducted rather efficiently,[4] the more realistic process of homomorphic matching requires the investigation of large numbers of partial subgraphs of the map for possible equivalency to pattern symbology.



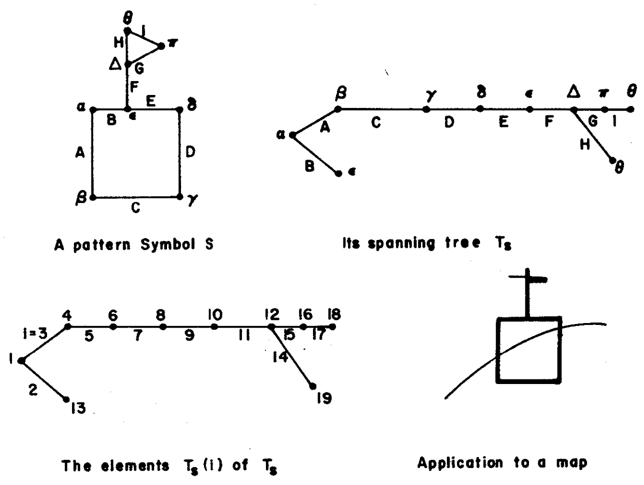
(a) a feature space

(b) Region containing instances of pattern symbology

(c) region formed by bounds testing of features

(d) best conservative region formed by bound testing of features

Figure 7—Partitioning of feature space by metric tests
(a) A feature space
(b) Region containing instances of pattern symbology
(c) Region formed by bounds testing of features
(d) Best conservative region formed by bound testing of features



A pattern Symbol S

Its spanning tree T_s

The elements $T_s(I)$ of T_s

Application to a map

Figure 8—The structure of a pattern symbol

In order to limit the number of partial subgraphs which need be checked for homomorphic match, metric equivalency tests have been integrated into the graph theoretic isolation process. These tests include bounds checking of the lengths, curvatures, thicknesses, and angles between lines, and may be easily extended as required.

If the metric tests are well chosen then they will be conservative, i.e., will not reject any true instance of pattern symbology. This may be seen by viewing the various screening quantities as features in the feature space of Figure 7. The ensemble of all true instances of pattern symbology will form some region *A* in the space, Figure 7. Any set of metric tests may also be viewed as partitioning the feature space, passing only those instances of symbology which lie in some region *B* of the space formed by the partition. If region *B* contains region *A*, then the set of tests is conservative. If region *B* exactly contains region *A*, then the set of tests also form a perfect recognizer. It is more important however, that the tests result in a high processing efficiency. This may be achieved by immediate testing of each feature as it is first calculated. This form of testing generates a partition which boxes in some region of feature space as shown in Figure 7c. While this partition is not necessarily perfect, it is usually possible to adjust the bounds of the tests so as to achieve a near optimal, as well as conservative, performance on the basis of limited sampling of pattern symbology within a map, Figure 7d. Thus the isolation process may also often serve well as the final stage of recognition. When desired, however, it is always possible to concatenate other more conventional recognition processes in order to achieve yet higher accuracies of recognition.

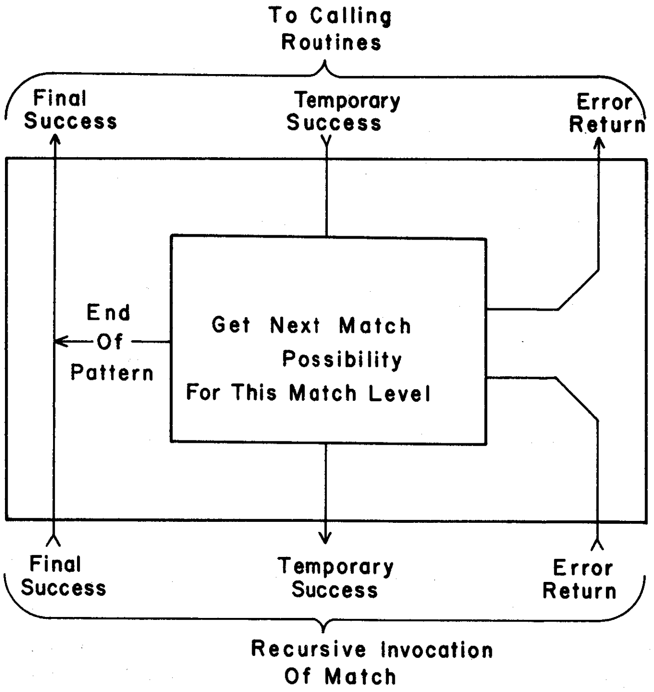


Figure 9—Structure of MATCH

The routine MATCH

Application of the search for graphical and metric equivalencies is conducted via a recursive routine called MATCH. On the graph-theoretic level, MATCH functions through utilization of tree programming. In this approach, a spanning tree T_s is pre-specified for each pattern symbol S. The elements of T_s are named $T_s(i), i=1, 2, \ldots, N_s$, where $T_s(i)$ is constrained to be connected to $T_s(1)$ through the set $\{T_s(j), j=1, 2, \ldots, i\}$. These structural conventions, illustrated by Figure 9, are developed to permit utilization of a recursive search policy in matching T_s and the partial subgraphs of a map G_m.

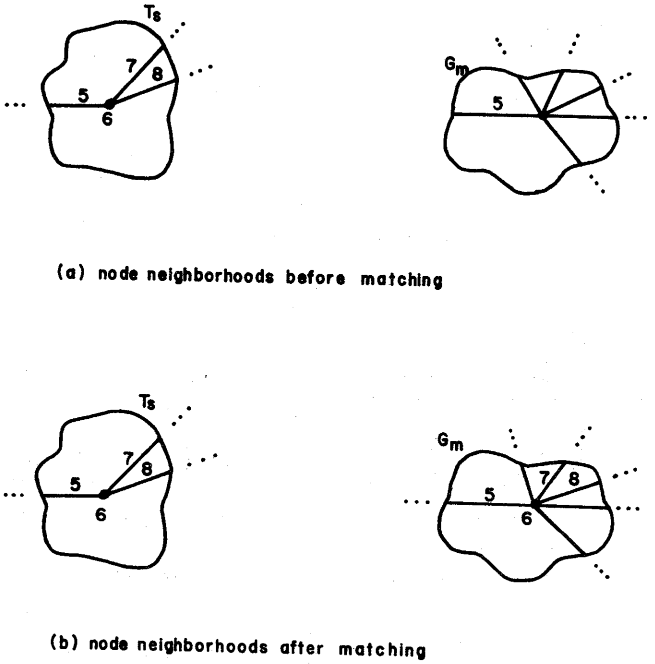
The recursive structure of MATCH is shown in Figure 9. It has one entry from and two exits back to the calling program. Being recursive, it can call itself. At the ith level of recursion, MATCH investigates the possibilities of homomorphic equivalence of elements G_m to $T_s(i)$. As each possibility is proposed MATCH checks to insure that all implied graph-theoretic equivalencies between $T_s(j), j=1, 2, \ldots, i$, and G_m are acceptable, and that basic metric equivalences are met.

More explicitly, at the recursive level i, MATCH takes the following action. If T_s has been fully matched then MATCH takes a *FINAL SUCCESS* exit which carries it back up the recursive string with the isolated symbology from G_m. If all matching possibilities for $T_s(i)$, $i>1$, have been exhausted then MATCH takes an ERROR RETURN exit back to the $i-1$th level of recursion in order to try to find other matching possibilities for $T_s(i-1)$. Alternatively if all matching possibilities for $T_s(i)$, $i=1$, have been exhausted then MATCH fails to isolate the symbology sought and exits along the ERROR RETURN exit to the calling program. On the other hand, if it finds an acceptable match for $T_s(i)$ then it exits via the TEMPORARY SUCCESS exit to continue the matching search for $T_s(i+1)$.

At each recursive level, MATCH performs one of three specific actions: matching to nodes of T_s, matching to lines of $T_{s'}$ and initial matching to new pattern components of T_s.

Matching of nodes

The fundamental operation performed by MATCH is the matching of the immediate neighborhood of a node of G_m to that of T_s. This matching must satisfy several constraints. It must be feasible, must satisfy



(a) node neighborhoods before matching

(b) node neighborhoods after matching

Figure 10—Matching of node neighborhoods
(a) Node neighborhoods before matching
(b) Node neighborhoods after matching

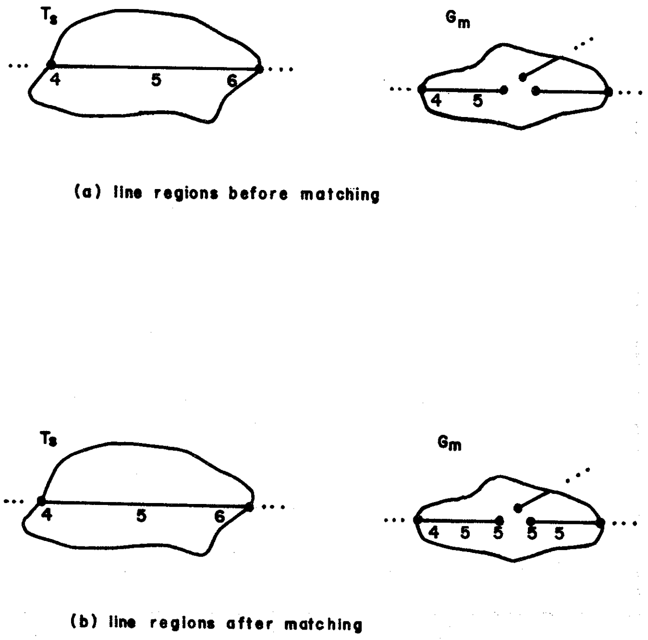

Ts
Gm
4
5
6
4
5
(a) line regions before matching
Ts
Gm
4
5
6
4
5
5
5
5
(b) line regions after matching

Figure 11—Matching of line regions
(a) Line regions before matching
(b) Line regions after matching

certain angular conditions, and must not violate any prior matching assumptions. A matching is feasible if the degree of the node of G_m is greater than or equal to that of the node of T_s. This requirement, for example, results in termination of the matching of the map segment of Figure 8 at recursive stage 16 because the degree of node $T_s(16)$ was greater than the degree of the corresponding node of G_m.

A matching satisfies necessary angular constraints if all internal angles of the planar graphs of G_m and T_s are sufficiently similar. It satisfies prior matching assumptions if the present matching attempt is not in conflict with previous matching attempts or involves lines of G_m which are matched to other pattern symbology.

The neighborhood of a node of T_s is considered to be fully matched when the node and its adjacent lines are matched to a node of G_m and some subset of its adjacent lines. For instance, if the conditions represented in Figure 10a hold upon a call of MATCH, then Figure 10b shows a suitable match of the neighborhoods.

Matching of lines

In matching a line of T_s to elements of G_m, MATCH finds a path in G_m which corresponds to the line of T_s, This path may contain one or more elementary lines and may even contain breaks. The path must, however, satisfy minimal constraints. It must not cross over itself, no portion of the path other than endpoints may have been previously matched, no breaks may be adjacent, the implied endpoint matchings must be consistent with prior matchings, and finally certain metric equivalencies must be observed. Typically these metric equivalencies need be no more complex than a rough correspondence of length and curvature between line of T_s and the path within G_m.

As an example of the matching of lines, consider Figure 11. If the conditions of Figure 11a hold upon a call of MATCH then Figure 11b shows a suitable matching between the line of T_s and a path within G_m.

Initial matching of components

Matching of nodes and lines of connected symbology is conducted by the tracking of connectivity via T_s. This technique may be extended to the matching of symbology S composed of disjoint components through inclusion of lines within T_s which link nodes of the various components of S. These lines may, for matching purposes, be treated as straight lines of T_s, thereby simplifying the matching process.

FINAL CLASSIFICATION

MAPPS has the capability for inclusion of a final classification routine (CLASS). When used this routine serves to provide a final decision as to whether an isolated piece of symbology is a true instance of the symbology being sought. If the isolation is determined to be erroneous, then MATCH continues its search toward an alternative isolation.

CLASS may be implemented by a variety of approaches, the best known and most appropriate of which is through use of discriminant functions. The power of its application can be dramatically enhanced through proper use of results from MATCH. For example, quite complex features can be devised for input to CLASS from the very detailed description of the isolated symbology produced by MATCH. As further example MATCH may be used to isolate new symbology and tentative classification from a map to form a training set for CLASS. Then with or without a teacher, the discriminant function underlying CLASS can be perturbed toward a more optimal setting by any of several well-known algorithms.[3]

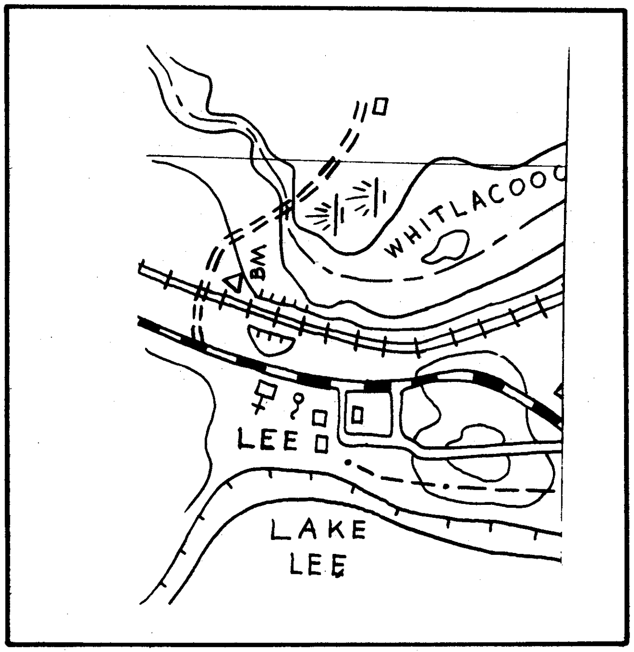


Figure 12—Test results
Figure 12a—The input map

OUTPUT

The Output Routine (OUTPUT) takes the isolated symbology recognized by earlier routines (MATCH, CLASS, . . .) and produces the final MAPPS output in accordance with a specific user query. This is accomplished by establishing a data structure in which data can be retrieved through use of relational pointers. Retrieval is effected by specification of the desired symbology class and by calling various relations. The relation "contains" may be used, for instance, to find

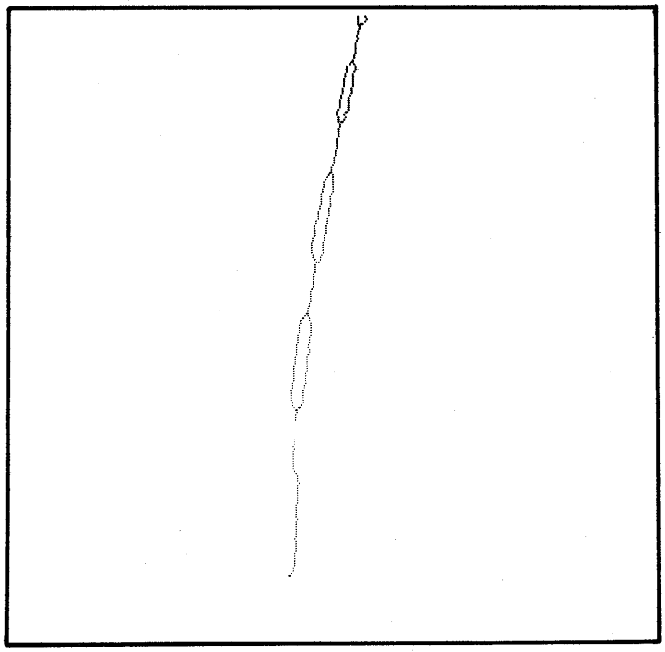


Figure 12c—Isolated highways

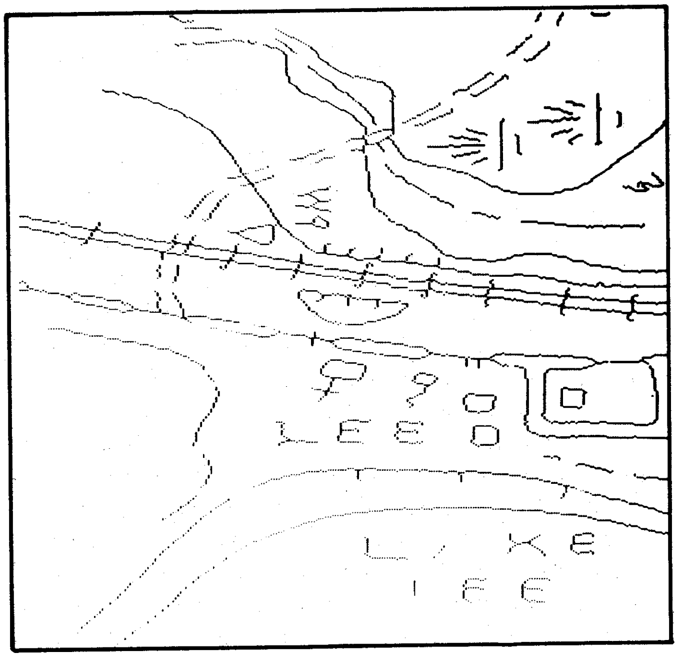


Figure 12b—The input map after line extraction

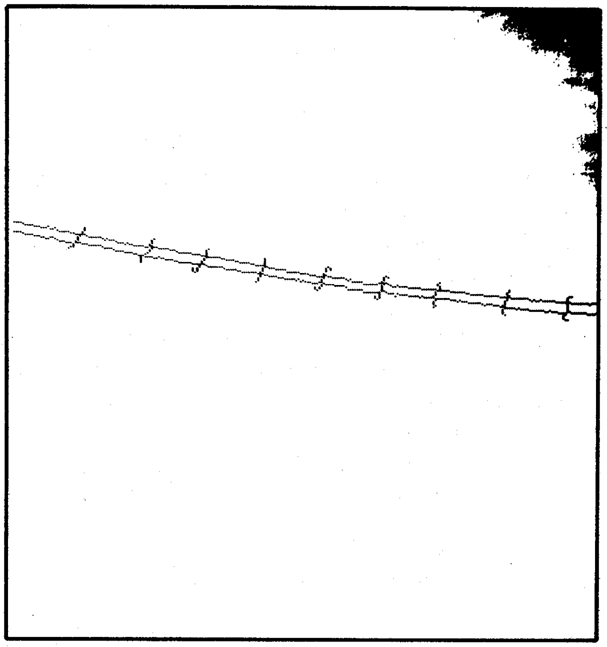


Figure 12d—Isolated railways

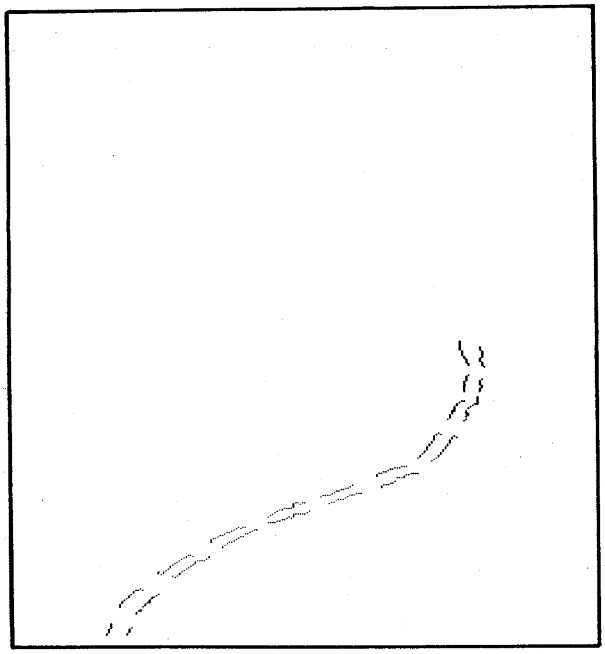


Figure 12e—Isolated roads

the various isolated symbols belonging to a specified symbology class. Another call of "contains" will then result in presentation of all lines and nodes present in the specified symbology. Yet another call of "contains" will return the specific picture points involved. Alternatively a call of "position" will return the nominal location of the center of each isolated symbology.

RESULTS OF TEST RUNS

MAPPS has been tested on several map regions. In each case CLASS was set to accept all isolations in order to most stringently test the operation of MATCH. Throughout all testing the results were highly satisfactory. Figure 12 presents the results for a representative run. The map region of Figure 12a was fed to the early stages of MAPPS producing the preprocessed map of Figure 12b. This preprocessed map was then subjected to several searches for specified symbology resulting in Figures 12c through 12k. In all but one case the recognition was conservative. Only in the case of Figure 12f was a false isolation made. An *M* was there recognized as an *E*. Had CLASS been implemented using character recognition techniques, this misrecognition could have been avoided. In those cases where recognition was incomplete, as for the highway of Figure 12c, isolation was terminated by MATCH due to mismatch of structure between the map and symbology sought.

Some overall statistics on the test run: MAPPS correctly found instances of 18 types of lineal and point symbologies. These instances were formed from 5382 elementary lines. In addition 7 incorrect instances were

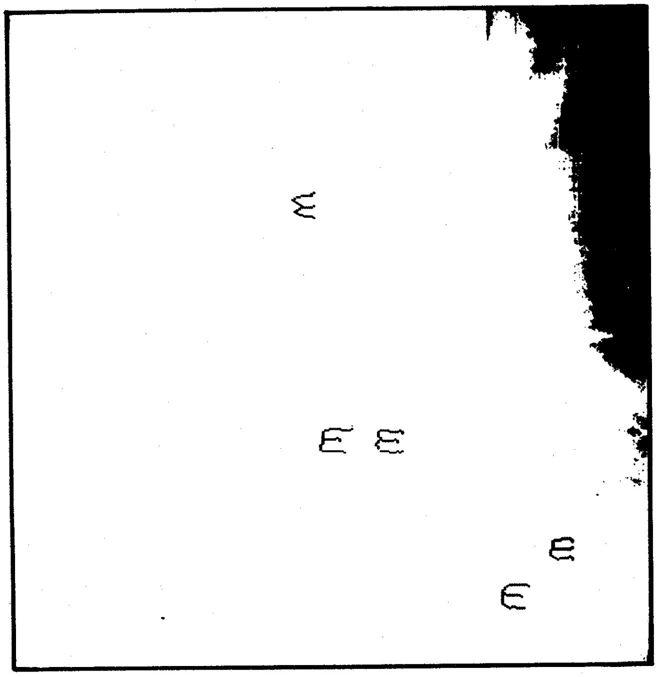


Figure 12f—Isolated 'E's

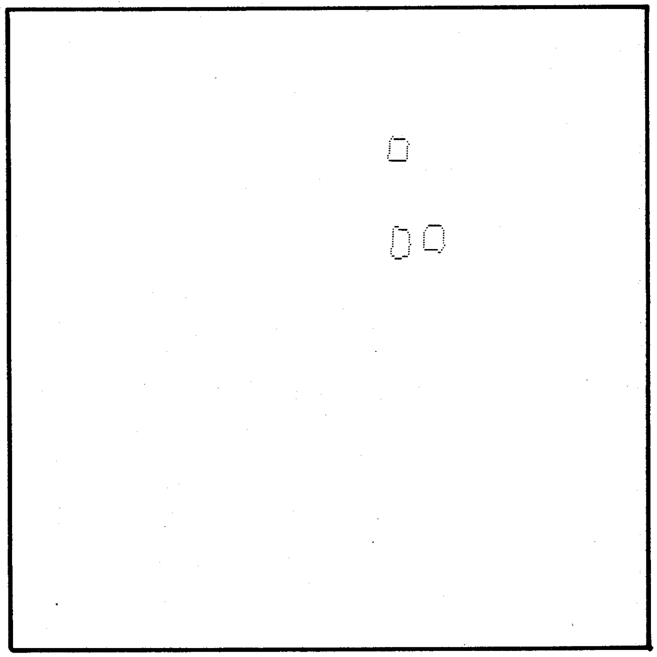


Figure 12g—Buildings

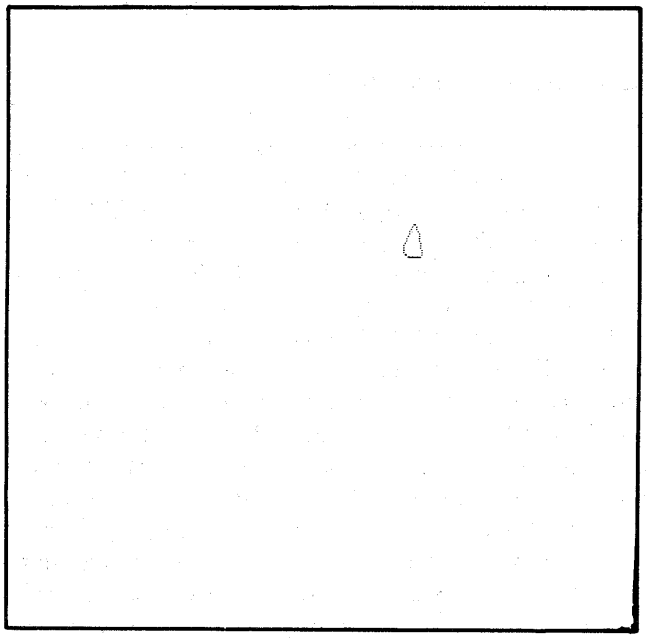


Figure 12h—Benchmark symbols

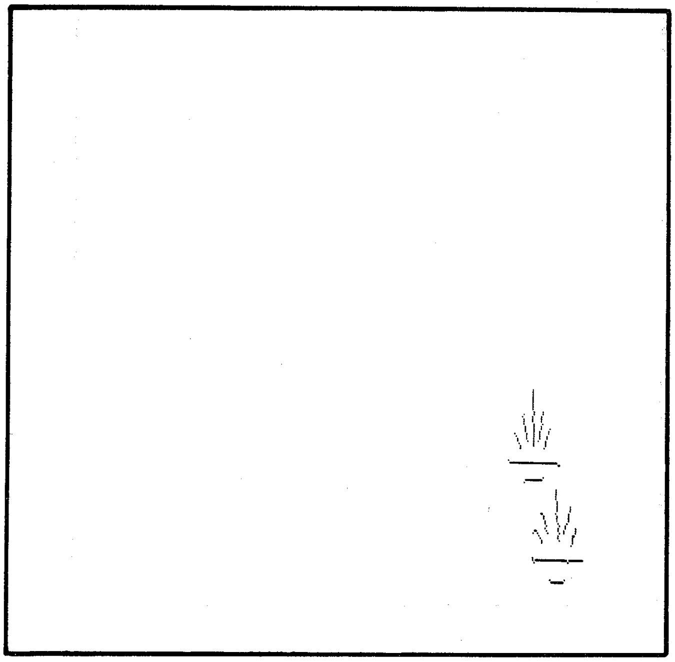


Figure 12j—Swamp symbols

isolated although in each case this could have been avoided by use of a proper classification structure within CLASS. Since minimization of run-time was of minor importance, the average test-run for each symbology search of a map region took approximately 10 minutes from input film to output description. It is estimated that this could have been improved very significantly by various means; however this was not a major goal at this stage of research.

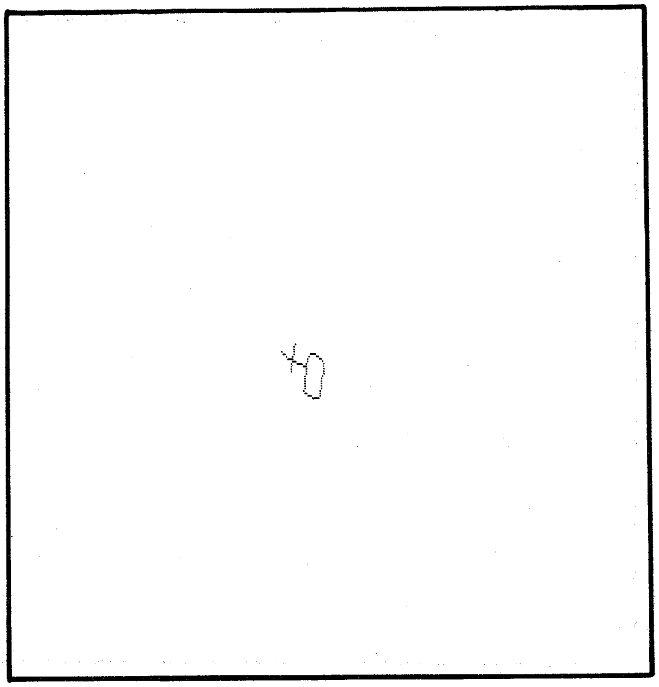


Figure 12i—Churches

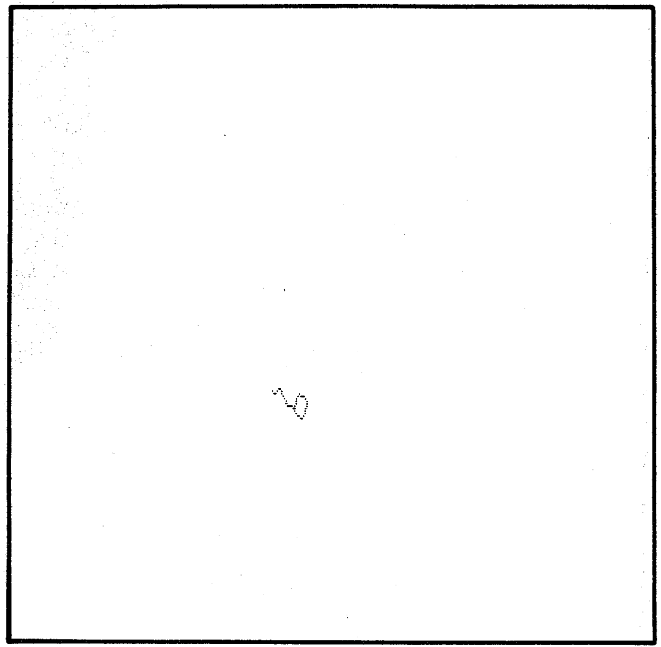


Figure 12k—Spring symbols

CONCLUSION

This work has been an investigation into a broad class of conventionalized, two-dimensional, pictorial patterns: the symbology of maps. Important aspects of the problem involve line extraction, isolation under conditions of qualitatively-defined degradation, use of graph structures and matching techniques in isolation, and interactive recognition of geometrically variable symbology.

A sophisticated approach to line extraction yielded a useful data base upon which to conduct symbology isolation and recognition. The use of graph structure and matching in symbology isolation proved very effective. Unexpectedly, it was found to be seldom necessary to resort to formal classification techniques in recognition of the isolated symbology. Such techniques could be incorporated as desired resulting in a continued search for symbology in case of any misisolation. The program as a whole is able to be expanded to the recognition of a wide variety of graphical symbology. In addition, the concepts involved can quite possibly be applied to the automated perception of gray-level sceneries such as blood cells, aerial photographs, chromosomes, and target detection.

ACKNOWLEDGMENT

The authors would like to acknowledge the interest displayed by other members of the Center for Informatics Research in this and related research. This research has been sponsored in part by the Office of Naval Research under Contract No. N00014-68-A-0173-0001, NR 049-172.

REFERENCES

1 R H COFER
Picture acquisition and graphical preprocessing system
Proceedings of the Ninth Annual IEEE Region III
Convention Charlottesville Virginia 1971

2 R H COFER J T TOU
Preprocessing for pictorial pattern recognition
Proceedings of the NATO Symposium on Artificial
Intelligence Rome Italy 1971

3 J T TOU
Engineering principles of pattern recognition
Advances in Information Systems Science Vol 1 Plenum
Press New York New York 1968

4 G SALTON
Information organization and retrieval
McGraw-Hill Book Company New York 1968

Computer generated optical sound tracks

by E. K. TUCKER, L. H. BAKER and D. C. BUCKNER

University of California
Los Alamos, New Mexico

INTRODUCTION

For several years various groups at the Los Alamos Scientific Laboratory have been using computer generated motion pictures as an output medium for large simulation and analysis codes.[1,2,3] Typically, the numerical output from one simulation run is so large that conventional output media are ineffective. The time-variable medium of motion picture film is required to organize the results into a form that can be readily interpreted. But even this medium cannot always convey all of the information needed. Only a limited number of variables can be distinctly represented before the various representations begin to obscure or obliterate each other. Furthermore, the data presented usually must include a significant amount of explanatory material such as scaling factors, representation keys, and other interpretive aids. If a film is to have long-term usefulness to a number of people, this information must either be included on the film or in a separate writeup that accompanies the film.

In an effort to increase the effective information density of these films, a study was undertaken to determine the feasibility of producing optical sound tracks as well as pictorial images with a microfilm plotter. Some exploratory work done at the Sandia Laboratories, Albuquerque, New Mexico, suggested that this might provide a good solution to the problem.[4] It has been demonstrated many times that a sound track facilitates the interpretation of visual presentations.[5] However, from our standpoint, the addition of another channel for data presentation was as important as facilitating interpretation. Not only could a sound track present explanatory and narrative material efficiently and appealingly, it could also be used to represent additional data that might otherwise be lost. For example, it is always difficult to clearly represent the movement of many particles within a bounded three-dimensional space. If, however, the collisions of particles—either with each other or with the boundaries of the space—were represented by sounds, interpretation of results would be greatly facilitated. This is feasible only if the sound track is computer produced, not "dubbed in" after the fact. It should be made clear at this point that it was not an objective of this project to have the computer create all of the waveforms represented on the sound track. What was required was that the computer be able to reproduce on an optical sound track any recorded audible sound, including voices or music. The waveforms that the computer would actually have to create could be limited to some of the sounds we wanted to use as data representations.

OPTICAL SOUND TRACKS

Sound is generated by a vibrating body which produces a series of alternating compressions and rarefactions of some medium, i.e., a wave. As this series is propagated through the medium, particles of the medium are temporarily displaced by varying amounts. We shall speak of the magnitude and direction of this displacement as the instantaneous amplitude of the wave. If the variation of this amplitude can be described as a function of time, a complete description or encoding of the wave is obtained. Thus, a sound wave can be "stored" in various representations, as long as the representation fully describes the variation of amplitude with respect to time.

An optical sound track is one way of representing a sound. It consists of a photographic image which determines the amount of light that can pass through the track area of the film at a given point. As the film is pulled past the reader head, varying amounts of light pass through the film to strike a photocell, producing a proportionally varying electrical signal. A given change in signal amplitude can be produced at the photocell by varying either the area or the intensity of exposure of the sound track image.

Conventional sound tracks are produced by either of two methods. The variable area type of track is produced by having a beam of light of constant intensity pass through a slit of variable length to expose the film. In the variable intensity recording method, either the light's intensity or the slit width can be varied with the slit length held constant. Commercial sound tracks are produced by both methods. In both cases, the sound track image is produced on a special film that is moved past the stationary light source. Separate films of sound track and pictures are then reprinted onto a single film.

Sixteen-millimeter movies with sound have sprocket holes on only one edge. The sound track is located along the other edge of the film (see Figure 1). Such sound tracks are normally limited to reproducing sound with an upper frequency of 5000-6000 Hz. This limitation is imposed by the resolution that can be obtained with relatively inexpensive lens systems, film and processing and by the sound reproduction system of most 16 mm projectors.[6]

INPUT SIGNALS

In order not to be limited to the use of computer created sounds alone, it was necessary to be able to store

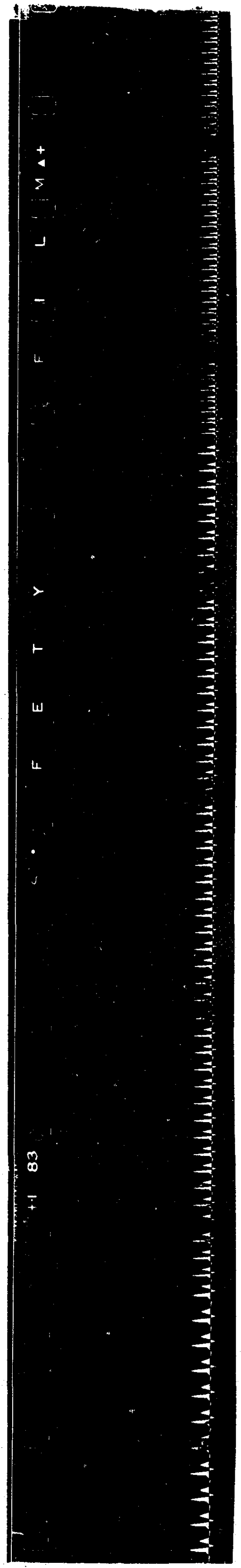


Figure 1—A computer generated optical sound track

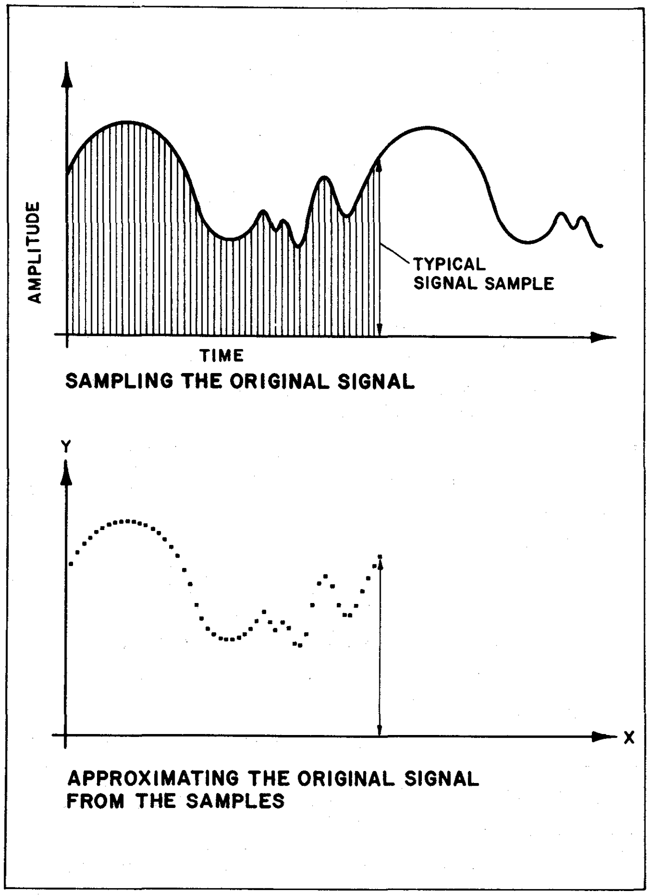

AMPLITUDE
TYPICAL
SIGNAL SAMPLE
TIME
SAMPLING THE ORIGINAL SIGNAL
Y
X
APPROXIMATING THE ORIGINAL SIGNAL
FROM THE SAMPLES

Figure 2—Discrete sampling

other complex audio signals, such as voices, in a form that could be manipulated by a digital computer. As discussed above, any audio signal can be completely described by noting the variation of the signal's amplitude as a function of time. Therefore, the data for a digital approximation of an audio signal can be obtained by periodically sampling the signal's amplitude (see Figure 2). The primary restriction associated with this approach requires that the sampling rate be at least twice the highest frequency contained in the signal.[7] In effect, samples obtained at a relatively low sampling rate *S* from a wave containing relatively high frequencies *f* will create a spurious "foldover" wave of frequency *S-f*.

The input for our experimental film was recorded on standard ¼-inch magnetic tape at a speed of 7½ IPS. Frequencies greater than 8000 Hz were filtered out, and the resulting signal was digitized at a sampling rate of 25,000 samples/second. The digitizing was performed on an Astrodata 3906 analog-to-digital converter by the Data Engineering and Processing Division of Sandia Laboratories, Albuquerque. The digital output of this process was on standard ½-inch 7-track digital magnetic tape in a format compatible with a CDC 6600 computer. This digital information served as the audio input for the sound track plotting routine.

PLOTTING THE SOUND TRACK

The sound track plotting routine accepts as input a series of discrete amplitudes which are then appropriately scaled and treated as lengths. These lengths are plotted as successive constant intensity transverse lines in the sound track area of the film. When these lines are plotted close enough together, the result is an evenly exposed image whose width at any point is directly proportional to an instantaneous amplitude of the original audio signal (see Figure 1). Consequently, as this film is pulled past the reader head, the electrical signal produced at the photocell of the projector will approximate the wave form of the original audio signal. The routine is written to produce one frame's sound track at a time. During this plotting, the film is stationary while the sound track image is produced line by line on the cathode ray tube of the microfilm plotter.

The sound reproduction system of a motion picture projector is very sensitive to any gaps or irregularities in the sound track image. Plotting a sound track, therefore, requires very accurate film registration. Furthermore, the sound track image must be aligned in a perfectly vertical orientation. If either the registration or the vertical alignment is off, the track images for successive frames will not butt smoothly together and noise will be produced.

PLOTTER MODIFICATIONS

All of our early experimental films were produced on an SD 4020 microfilm printer/plotter. Three modifications had to be made to the 16 mm camera of this machine in order to make these films. These modifications do not affect any of the camera's normal functions.

In the first modification, the Vought 16 mm camera had to be altered to accommodate single sprocketed 16 mm movie film. For this it was necessary to provide a single sprocketed pull-down assembly. This was accomplished by removing the sprocket teeth on one side of the existing double sprocket pull-down assembly. Next, it was necessary to replace the existing lens with a lens of the proper focal length to enable the camera to plot the sound track at the unsprocketed edge of the film. The lens used was a spare 50 mm lens which had previously been used on the 35 mm camera. With the existing physical mountings in the 4020, this 50 mm lens presents, at the film plane, an image size of approximately 17.5×17.5 mm. Thus, with proper raster addressing, a suitable 16 mm image and sound track may be plotted on film. (Increasing the image size in this fashion produces a loss of some effective resolution in the pictorial portion of the frame while the 50 mm lens is in use. This loss of resolution in the picture portion is not particularly penalizing in most applications.) Finally, it was necessary to expand the aperture both horizontally and vertically to allow proper positioning and abutment of the sound track on the film.

By interchanging the new lens with the original lens, normal production can be resumed with no degradation caused by the enlarged aperture and single sprocketed pull-down. No other modifications were required on the SD 4020 in order to implement the sound track option.

The primary difficulty we encountered using the SD 4020 was that we could not get consistently accurate butting of consecutive frames. Therefore, the later films were plotted on an III FR-80, which has pin registered film movement. In order to use this machine, the film transport had to be altered to accommodate single sprocketed film, and the aperture had to be enlarged. A software system tape was produced to allow the sound track image to be plotted at the unsprocketed edge of the film, with the pictorial images still plotted in the normal image space. The FR-80 also provides higher resolution capabilities, so that no loss of effective resolution is incurred when pictorial images and the sound track are plotted in one pass through the machine.

As was discussed earlier, optical sound tracks are usually limited to reproducing sound with an upper frequency of 5000-6000 Hz. Since motion picture film is projected at a rate of 24 frames/second, a minimum of

410 lines per frame are needed to represent such frequencies in the sound track. While we have made no quantitative tests to demonstrate the production of such frequencies, we would expect efficient resolution to produce frequencies in or near this range with either of the plotters. Our applications so far have not needed the reproduction of sounds in this frequency range.

THE TRACK PLOTTING ROUTINE

The present sound track plotting routine was written with three primary objectives in mind. First, it was felt that it would be advantageous to be able to produce both pictorial imagery and the sound track in one pass through the plotter, with the synchronization of pictures and sound completely under software control. Second, the routine was written to allow the user maximum flexibility and control over his sound track "data files". Finally, the routine was designed to produce film that could be projected with any standard 16 mm projector.

One-pass synchronization

The sound track plotting routine is written to produce one frame's sound track at a time, under the control of any calling program. However, in a projector, the reader head for the sound track is not at the film gate; it is farther along the threading path. The film gate and the reader head are separated by 25 frames of film. Therefore, to synchronize picture and sound, a frame of sound track must lead its corresponding picture frame by this amount so that as a given frame of sound track arrives at the reader head, its corresponding pictorial frame is just reaching the film gate. In order to be able to generate both picture and sound in one pass through the plotter, it was necessary to build a buffer into the sound track plotting routine. This buffer contains the plotting commands for 26 consecutive frames of film. In this way, a program plotting a pictorial frame still has access to the frame that should contain the sound track for the corresponding picture.

The simultaneous treatment of pictorial plot commands puts the synchronization of pictures and sound completely under software control. Furthermore, this can be either the synchronization of sound with picture or the synchronization of picture with sound. This is an important distinction in some applications; the current picture being drawn can determine which sound is to be produced, or a given picture can be produced in response to the behavior of a given sound track wave.

Flexibility

The present routine will read from any number of different digital input files and can handle several files simultaneously. Thus, for example, if one wishes to have a background sound, such as music, from one file behind a narrative taken from another file, the routine will combine the two files into a single sound track. The calling routine can also control the relative amplitudes of the sounds. In this way, one input signal can be made louder or softer than another, or one signal can be faded out as another one fades in. Any input file can be started, stopped, restarted or rewound under the control of the calling program.

DEMONSTRATION FILMS

Several films with sound have been produced using the sound track plotting routine. Most of the visual portions were created with very simple animation techniques in order to emphasize the information content added by the sound track. The films review the techniques employed for the generation of a sound track. No attempts have been made to rigorously quantify the quality of the sounds produced since no particular criterion of fidelity was set as an objective of the project. Furthermore, the sound systems of portable 16 mm projectors are not designed to produce high fidelity sound reproduction, since the audio portion will always be overlaid by the noise of the projector itself. For our purposes it was enough to make purely subjective judgments on the general quality of the sounds produced.

SUMMARY

The ability to produce optical sound tracks, as well as pictorial imagery, on a microfilm plotter can add a tremendous potential to computer generated movies. The sound medium can serve to enhance the visual presentation and can give another dimension of information content to the film. This potential cannot be fully exploited unless the sound track and the pictures can be plotted by the computer simultaneously. Under this condition, the input for the sound track can be treated by the computer as simply one more type of data in the plotting process.

The input for the sound track plotting routine discussed in this report is obtained by digitizing any audio signal at a suitable sampling rate. This digital information can then be plotted on the film like any other data.

Very few hardware modifications were made to the plotter in order to produce sound tracks. The modifications that were made did not affect the plotter's other functions.

The routine is written to give the user as much flexibility and control as possible in handling his sound track data files. Multiple files can be combined, and synchronization is under the control of the user's program.

It now appears that the production of computer generated optical sound tracks will prove to be cost effective as well as feasible. If so, this process could conveniently be used to add sound to any computer generated film.

ACKNOWLEDGMENTS

While many individuals have made significant contributions to this project, the authors would like to give particular thanks to Jerry Melendez of Group C-4 for many hours of help in program structuring and debugging.

The work on this project was performed under the auspices of the U. S. Atomic Energy Commission.

REFERENCES

1 L H BAKER J N SAVAGE E K TUCKER
Managing unmanageable data
Proceedings of the Tenth Meeting of UAIDE Los Angeles California pp 4-122 through 4-127 October 1971

2 L H BAKER B J DONHAM W S GREGORY E K TUCKER
Computer movies for simulation of mechanical tests
Proceedings of the Third International Symposium on Packaging and Transportation of Radioactive Materials Richland Washington Vol 2 pp 1028-1041 August 1971

3 *Computer fluid dynamics*
24-minute film prepared by the Los Alamos Scientific Laboratory No Y-204 1969

4 D ROBBINS
Visual sound
Proceedings of the Seventh Meeting of UAIDE San Francisco California pp 91-96 October 1968

5 W A WITTICH C F SCHULLER
Audio visual materials
Harper & Row Publishers, Inc. New York 1967

6 *The Focal encyclopedia of film and television techniques*
Focal Press New York 1969

7 J R RAGAZZINI G F FRANKLIN
Sampled-data control systems
McGraw-Hill Book Company New York 1958

Simulating the visual environment in real-time via software

by RAYMOND S. BURNS

University of North Carolina
Chapel Hill, North Carolina

INTRODUCTION

Computer graphics has been seen since its inception[1] as a means of simulating the visual environment. Ivan Sutherland's binocular CRTs was the first apparatus designed to place a viewing subject in a world generated by a computer. When the subject in Sutherland's apparatus turned his head, the computer generated new images in response, simulating what the subject would see if he really were in the 3-space which existed only in the computer's memory. This paper describes a system which is a practical extension of Sutherland's concept.

The problem of simulating the visual environment of the automobile driver has attracted a variety of partial solutions. Probably the most used technique is simple film projection. This technique requires only that a movie camera be trained on the highway from a moving vehicle as it maneuvers in a traffic situation. The resulting film is shown to subjects seated in detailed mock-ups of automobile interiors, who are directed to work the mock-up controls to "drive" the projected road. The illusion of reality breaks down, however, when the subject turns the steering wheel in an unexpected direction and the projected image continues on its predefined course. Mechanical linkages from the mock-up to the projector, which cause the projector to swing when the steering wheel is turned, have also been tried. But that technique still breaks down when the subject chooses a path basically different from the path taken by the vehicle with the movie camera.

Such film simulators are termed "programmed". That is, what the subject sees is a function, not of his dynamic actions, but of the actions taken at the time the film was recorded. An "unprogrammed" simulator reverses this situation in that the image that the subject sees is determined only by his behavior in the mock-up.

Unprogrammed visual environment simulators have been built for studying driving behavior. The U. S. Public Health Service at the Injury Control Research Laboratory, Providence, Rhode Island, has constructed several examples of unprogrammed simulators. One of these features a model terrain board with miniature roads and buildings over which a television camera is moved through mechanical linkages to the steering wheel of an automobile mock-up. The television camera is oriented so that the subject is presented with a windshield view. This arrangement earns the "unprogrammed" label within the physical limits of the terrain board. In practice, however, its value as a research tool is limited to studying driver behavior at dusk, as the image presented to the subject is dim. Natural daylight illumination, even under cloudy conditions, is much brighter than the usual indoor illumination. Duplicating the natural daylight illumination over the surface of the whole terrain board was found to be impractical in terms of the heat produced and the current required by the necessary flood lamps.

Because of the difficulties and disadvantages of film- and terrain board-type simulators, some efforts in recent years have been directed toward constructing visual simulators based on computer-generated images. General Electric has developed a visual simulator for NASA, used for space rendezvous, docking and landing simulation, which embodies few compromises.[2] The G. E. simulator output is generated in real time and displayed in color. However, from a cost standpoint, such a simulator is impractical for use as a highway visual simulator because the G. E. simulator was implemented to a large extent in hardware.

Consequently, the search for a visual-environment simulator which could be implemented in software was initiated. A study, investigating the feasibility of such a simulator was undertaken by the Highway Safety Research Center, Chapel Hill, North Carolina, an agency of the State of North Carolina. This study led to the development of the VES, for Visual Environment Simulator, a black-and-white approximation of the GE-NASA spaceflight simulator, adapted for highway environment simulation and implemented in software.

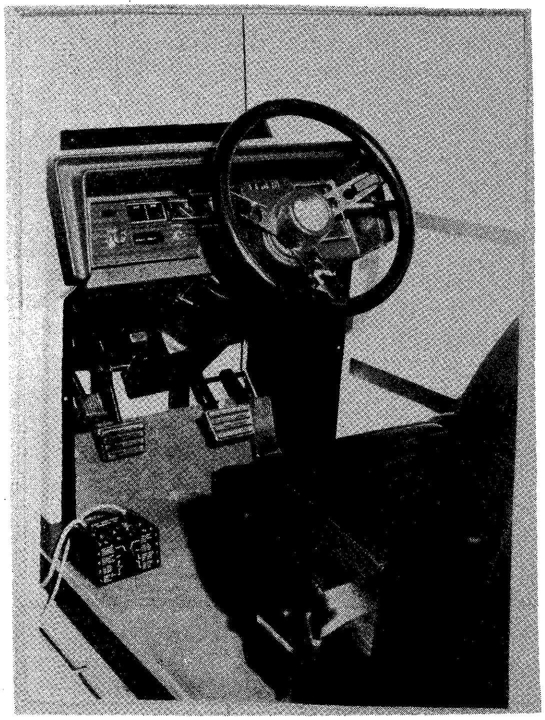


Figure 1—Mock-up of an automobile interior

VES DESIGN REQUIREMENTS

The requirements laid down by the Highway Safety Research Center were for a visual simulator that could be incorporated in a research device to totally simulate the driving experience to the subject. Not only was the visual environment to be simulated, but the auditory and kinesthetic environment as well.

The subject was to be seated in a mock-up of an automobile interior, complete with steering wheel, brake and accelerator (see Figure 1). The kinesthetic environment was to be simulated by mounting the mock-up on a moveable platform equipped with hydraulic rams. Under computer control, the mock-up could be subjected to acceleration and deceleration forces, as well as pitch, yaw and roll. Similarly, a prerecorded sound track would be synchronized with the visual simulation to provide auditory feedback. To as great a degree as possible, the subject was to be isolated from the real environment and presented only with a carefully controlled simulated environment.

From the researcher's point of view, this simulation device should allow him to place a subject in the mock-up, present him with a realistic simulated environment and then study the subject's reactions. Further, the choice of reactions available to the subject should not be limited in any way. So, if the subject were to "drive" off the simulated road and through the side of a simulated building, the visual, kinetic and auditory feedback should realistically reflect his actions.

A visual simulator to provide the feedback described above must meet several requirements. To support the subject's unlimited alternatives, each image generated by the visual simulator must be determined only by the subject's inputs via the steering wheel, accelerator and brake, together with the subject's position in the simulated terrain. Therefore, the entire image representing the visual environment must be calculated in the time span separating subsequent images.

REALISM

The high premium placed on realism in the visual simulator implied that the time span between subsequent images would be short, comparable to the time span between movie or television frames. The realism requirement also made hidden surface removal mandatory. Transparent hills, cars and road signs were unacceptable if the illusion of reality were to be maintained.

Further, television-type images were preferable to wire-frame drawings. If the images were to be of the wire-frame type, then objects would be represented by bright lines on the otherwise dark face of the CRT. For objects at close range, this representation presents few problems. But for objects at long range, the concentration of bright lines near the horizon would resemble a sunrise.

SYSTEM DESCRIPTION

The visual simulator software runs on a stand-alone IDIIOM-2 interactive graphics terminal consisting of a display processor, a VARIAN 620f mini-computer and a program function keyboard[3] (see Figure 2). The display processor is itself a computer, reading and executing its program (called a display file) from the core of the mini-computer on a cycle-stealing basis. The display processor's instruction set is extensive, but the visual simulator uses only a few instructions. Those used are instructions to draw horizontal vectors at varying intensities at varying vertical positions on the screen. The display processor is very fast, drawing a full screen (12") vector in about 20 microseconds. This speed allows a display file of seven thousand instructions to be executed in about $\frac{1}{30}$th of a second, effectively preventing image flicker at low light levels.

The VARIAN 620f mini-computer is also fast. Its core has a 750 nanosecond cycle time and most instructions require two cycles. Word size is 16 bits and core size is 16,384.

In its present configuration, the simulator receives its steering, braking and acceleration inputs from an arrangement of push buttons on the program function keyboard. The design configuration calls for the installation of an analog-to-digital converter and a driving station mock-up to replace the PFK. At the same time that the analog-to-digital converter is installed, a VARIAN fixed-head disk with a capacity of 128K words will be installed, giving the simulator nearly unlimited source data set storage.

The visual simulator (VES) accepts a pre-defined data set which describes a plan view of the terrain through which travel is being simulated. The terrain data set consists of (x, y, z) triples which describe the vertices of polygons.

At present, the VES input data set resides in the computer memory at all times. The main function of the VARIAN fixed-head disk mentioned above will be to store the VES input data set.

In operation, the VES system accesses a portion of the input data set corresponding to the terrain which is "visible" to the subject as a function of his position in the simulated landscape. Then, the steering, brake and accelerator inputs from the mock-up are analyzed and used to compute a wire-frame type view of the terrain which would be visible through a car's windshield as a result of such steering, braking or accelerating. Next, the hidden surface removal routine (HSR) processes each polygon to determine which polygons "obscure" others and to remove the parts of each that are obscured. The output of HSR is then converted into a program (display file) to be executed by the display processor. The display processor executed this program to draw the horizontal vectors at up to 8 different intensities which make up the television-like final image.

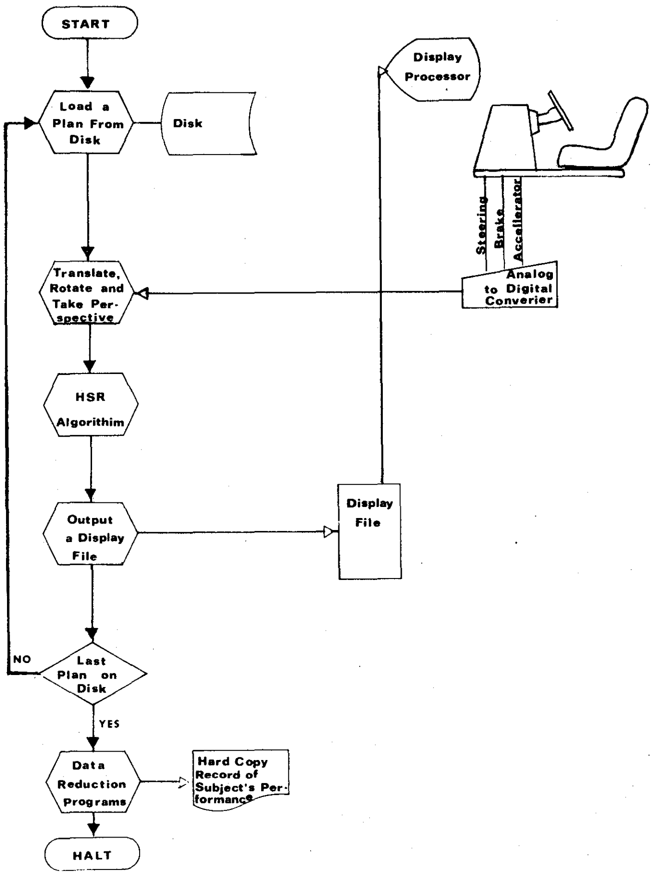


Figure 2—VES system block diagram

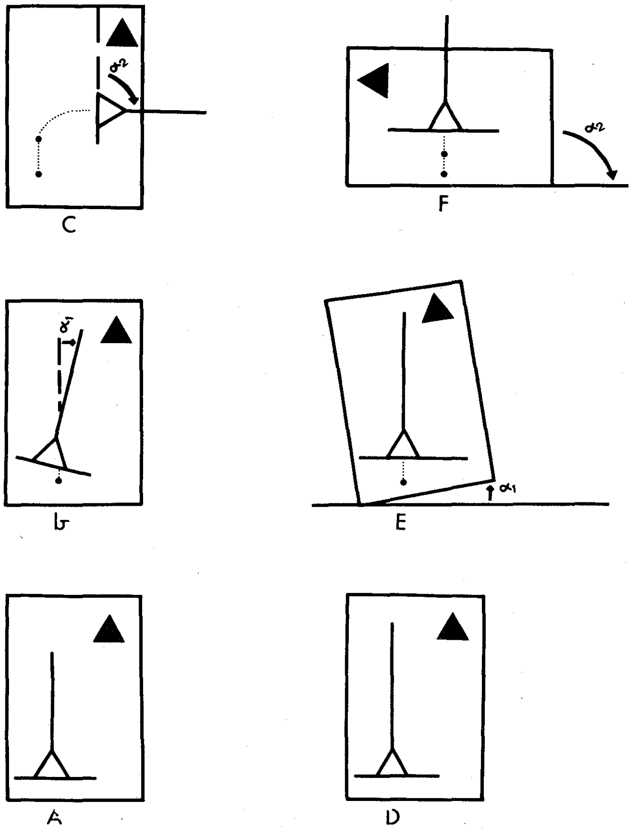


Figure 3—Diagram depicting subject's position (light triangle) moving through terrain data set versus data set moving past subject's position

The subject's position (see figure 3) in the terrain plan view is represented by the light triangle. The dark triangle represents a fixed object in the terrain. If the terrain is established as the frame of reference, the subject's position moves across the terrain. But from the point of view of the subject, who is stationary, the terrain must move toward him. The current angular position of the mock-up steering wheel in radians, relative to a fixed heading, is found in variable ALPHA. AL-

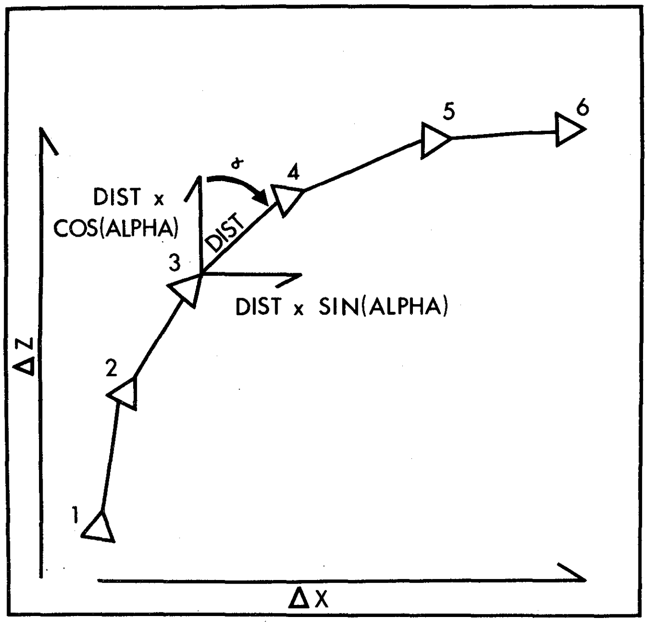


Figure 4—Detail of translation calculations

PHA is used to rotate the plan view represented in the input data set to give the effect of turning.

The current value of the forward speed of the subject's position is stored in variable DIST. DIST is the value resulting from combining the accelerator and brake inputs from the mock-up and is expressed in absolute distance units per frame.

To simulate the subject's motion through the terrain, ALPHA and DIST are used to rotate and translate the terrain data set relative to the subject's position. Figures 3, a, b, and c, depict the situation using the terrain data set as the frame of reference. Figures 3, d, e, and f reflect the same situation, interpreted in the subject's frame of reference. When the subject turns the steering wheel by α_1 radians clockwise, the terrain data set must be rotated by α_1 radians counter clockwise. And when the subject presses the accelerator or brake, the terrain data set is translated toward the subject by the value in DIST.

Because each image is computed relative to the initial terrain data set and not relative to the previous image, the translation step is more complicated than the rotation step. The amount that the terrain data set must be translated (DIST) must be decomposed into horizontal (X-coordinate) changes and depth (Z-coordinate) changes. Further, as the terrain data set must be translated from the original data set each time a new image is generated, the amount to be translated in either direction is a function of the "past history" of the subject's position (see Figure 4). For example, the correct amount to translate the terrain data set (hereinafter referred to by its name, PLAN) for position 5 depends upon previous position 4. That is, to translate PLAN to correspond to position 5 requires that the X values of each polygon in PLAN be reduced by ΔX and the Z values be reduced by ΔZ. ΔZ is computed by summing DIST × COS (ALPHA) for each value of DIST and ALPHA. Similarly, ΔX is the sum of DIST × SIN ALPHA.

After each polygon in PLAN has been translated and rotated, it is then operated upon by a perspective transformation to alter the plan view of the terrain to a perspective view with the view point placed at the subject's position.

PERFORMANCE

The image resulting from the VES is displayed on the face of the display processor, drawn on a raster of 512 horizontal lines having a resolution of 1,024 points per line. The display processor has a P4 type CRT phosphor, which is the same as used for black-and-white television. The image is as sharp as any television frame and comparable to a photograph (see Figure 5).

Because of the simplicity of the terrain data sets used to date, the images are more cartoon-like than life-like.

One minor distraction is the rendering of slanted lines. When slanted lines are nearly vertical, they are represented on the screen by many horizontal rasters and appear quite smooth. When a line is nearly horizontal, however, it is represented by only a few rasters. Consequently, the length of the vectors used to represent the slanted line vary widely from raster to raster. These large changes in vector length are easily detected by the observer as jagged slanted lines.

Image flicker, caused by the CRT phosphor darkening significantly between passes of the electron beam, is only a small problem. This happy situation is partly because of the low ambient light levels used when the VES is operating and partly because the display processor is very fast. A display of 12 polygons involves about 3,100 display commands which are executed easily in $\frac{1}{30}$th of a second, allowing each point in the picture to be intensified 30 times per second.

However, pictures of twice the present complexity will require roughly twice as long a display file. Under those conditions, the picture could not be intensified 30 times per second and image flicker would likely become a serious problem.

In a recent test run, involving a simple terrain data set, the VES took 30 seconds to produce 300 frames. This yields an average time per frame of about $\frac{1}{10}$th of a second which is slower than the frame rate of home movies. The illusion of continuous motion is main-

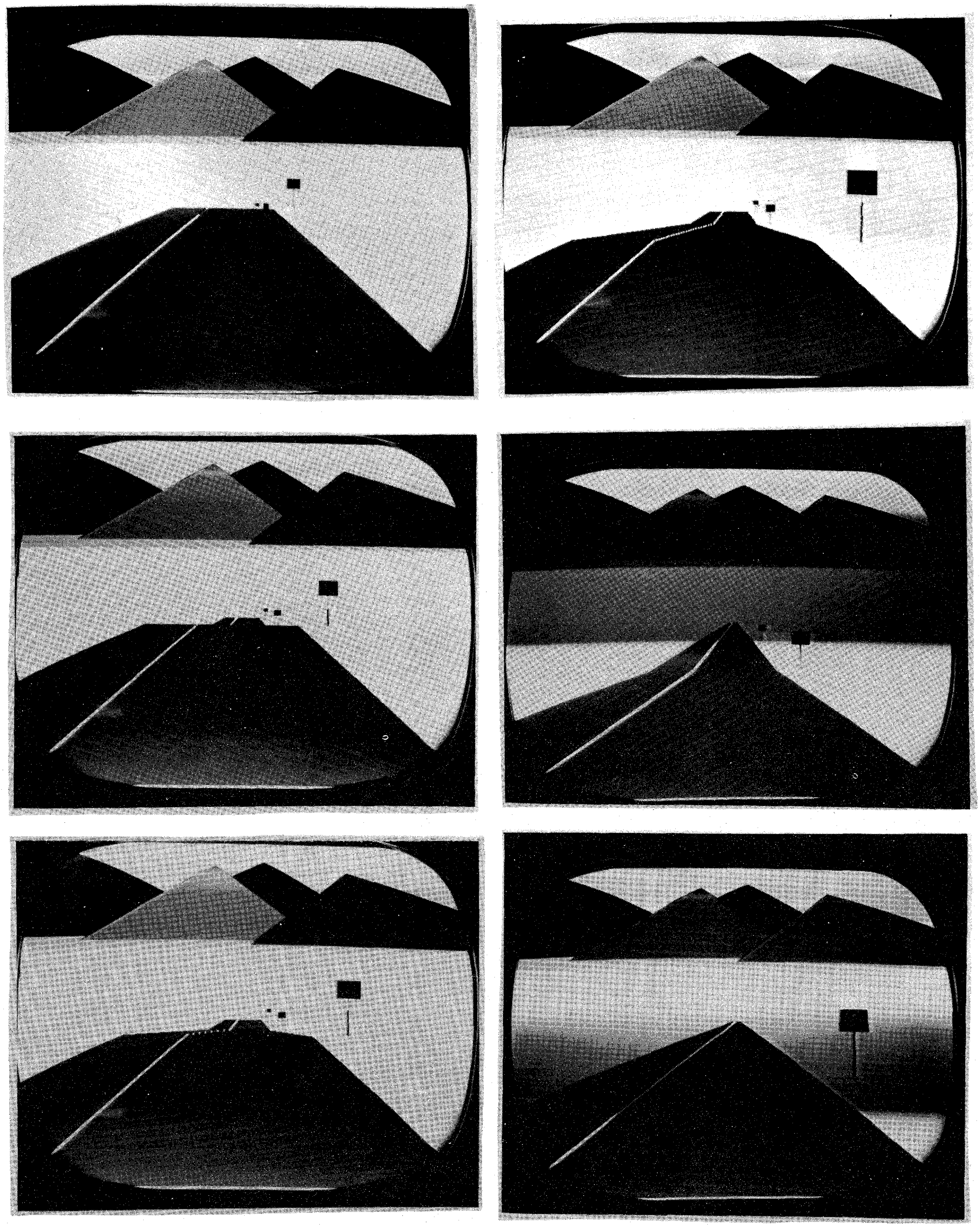


Figure 5—Sequence of still photographs from the VES in operation

tained, however, for all but the highest rates of angular velocity encountered in a typical highway environment. For example, a road sign in the distance appears to move smoothly. But as the subject's "car" draws abreast of the sign, the angular velocity becomes very high, reaching a maximum as the sign is "passed." In the frames just preceding the sign's disappearance at the edge of the screen, its motion becomes jerky. The situation is aggravated in that the generated images are sharp from edge to edge whereas the corresponding television image tends to be blurred when the angular velocity is large.

Despite these disadvantages, the VES simulates simple highway scenes sufficiently well for use as a practical research tool. Most driving situations can be abstracted so that only a few vehicles and road side objects are necessary to the representation. In fact, controlled experiments normally require a high degree of abstraction to spot light the particular aspect under study. Consequently, the VES is well adapted for the study of a wide range of driving behavior.

But simple scenes are not necessarily realistic scenes. And realism was the main point of the original simulator concept. In this light, the current VES is a rough approximation to the device described in the design specifications. To produce a visual environment simulator which does live to the original concept requires development of the present VES.

FUTURE DEVELOPMENT AND IMPROVEMENT

As was mentioned earlier, the design configuration includes a fixed-head disk, an analog-to-digital converter and a driving station mock-up. The first stage of improvement and development will involve integrating these devices into the system when they are installed.

By designating some polygons as "independent" and assigning speed and direction parameters to these polygons different from those input by the subject, vehicles can be made to maneuver on the screen independently of the subject. The researcher conducting the experiment can steer these vehicles along the same "highway" the subject sees and present him with a wide range of driving situations in which the subject must interact with other vehicles. This "independently moving vehicle" feature (IMV) is partially implemented. What remains is to devise a steering means for the researcher, along the lines of a miniature mock-up, to allow him to control his vehicle on the screen.

An interactive program to allow the automated generation of the input terrain data set is also being developed. This program (GIPC) will permit the researcher to design and preview a driving course before it is presented to the subject. To design an input terrain data set, the researcher loads GIPC from the disk. When GIPC is running, the screen is roughly divided into thirds (see Figure 6). The upper left third initially presents a menu of objects such as roads, trees, signs, etc. The bottom left third is a slide rule-like scale. A small box, just above the scale, contains the number selected from the scale by a light pen hit. The upper right third of the screen is a plan view of the terrain being designed while the box in the lower right corner contains program control commands which are activated by light pen hits.

In use, the researcher selects an item (say trees) from the menu via a light pen hit. That portion of the screen then changes to display the available selection of trees. On selecting the desired form of tree, the screen indicates GIPC is waiting for the size parameter for the selected tree. This parameter is input by sliding the light pen down the scale and observing the number displayed in the box above the scale. When that number is the one desired, it is selected by flicking the light pen away from the scale. When that is done, the object chosen is displayed in the plan view in the upper right box. The menu returns to the box in upper left to await the next selection. When the desired complexity has been built into the terrain data set, it is stored onto disk via a light pen hit in the program control box.

Another feature desirable for practical use of the VES is data reduction facilities. Various parameters describing the subject's behavior are available to be measured and recorded during an operational run. In future development, these will be retained on the disk

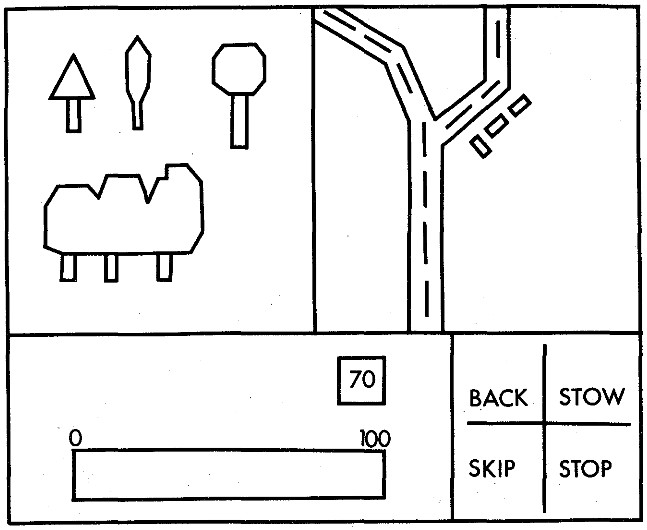


Figure 6—GIPC screen layout

and analyzed statistically after the run has ended, leaving only the results of the statistical programs to be printed out for each subject.

APPENDIX

The hidden surface removal algorithm (HSR) is a development and specialization of an alogrithm due to W. Jack Bouknight. An outline of that alogrithm follows. Readers are referred to the reference for a detailed discussion of LINESCAN.[4]

The image resulting from LINESCAN is made up of 512 horizontal rasters or scan lines. Picture shading is accomplished by varying the intensity of the CRT beam at appropriate points along the length of the raster.

Bouknight's approach to removing hidden surfaces is straightforward. His algorithm accepts a data set of projected polygons and decomposes the polygons into a chain of points ordered on increasing Y values of the points. Each point is tagged with its polygon number and the addresses of its neighboring points. The data set containing this chain of points is accessed once for each scan line to select all the points which correspond to a single raster position. These are simply the points with Y values equal to the current raster position.

To process a single scan, the set of points with Y values common to the Y value of the current raster are formed into a table referred to as the active line table or ALT. Logically, the ALT contains the intersection points of each polygon in the input data set with the current scan line. For example, in Figure 7, the points of intersection with scan line "a" make up the current elements in the ALT. These elements are examined sequentially from left to right in the following manner. Each polygon has an associated flag which is kept in a table and altered to signal the presence of the scan. On encountering point "a", the flag for the triangle is set to "in" to indicate that the scan has "entered" the triangle. A search is made of the table of flags and as no other flags are found set to "in," LINESCAN outputs the ordered triple (X_1, Y_1, PN_t) where X_1 is the X value of point a_1, Y_1 is the Y value of the scan line and PN_t, the polygon number of the triangle. On encountering a_2, LINESCAN sets the flag for the rectangle indicating that the scan is "in" the rectangle. Now two flags are set to "in" and a "depth sort" is required to determine which polygon is behind the other. That is, the Z-depth of the rectangle and the triangle are compared. As indicated by the dashed lines, the rectangle is "behind" the triangle. In this case, no ordered triple is output. On encountering a_3, the triangle's flag is set "out". As there is now only one polygon with flag set to "in", the ordered triple (X_3, Y_3, PN_r) is output. Similarly, as a_4 is encountered, the rectangle's flag is set to "out" and the triple (X_4, Y_4, PN_r) is output. This concludes LINESCAN's processing of a single scan line.

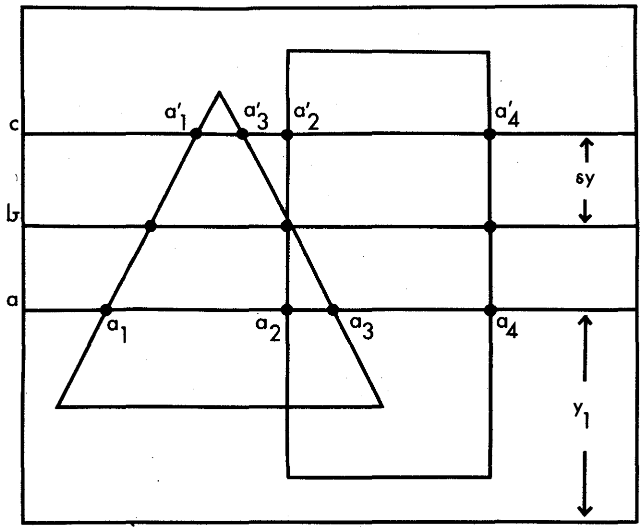


Figure 7—Detail of LINESCAN operation

To obtain the set of intersections corresponding to scan line "b," each element in the ALT for scan line "a" must be modified by an amount determined by the space between raster elements, δY, and the slope of the polygon's face. Because polygons are composed of straight line segments, the change necessary is constant for each given line segment. To obtain the ALT entries for scan line "b," this constant value is added to the previous entries in the ALT.

But before processing scan line "b" can begin, the new ALT is re-sorted on increasing X values. This step is required because when the new ALT is constructed from the old by the addition of the slope constants mentioned above, the order of some points may be disturbed. Note that this situation occurs when the ALT for scan line "c" is generated. Because of the differing slopes of the triangle and rectangle sides, a_3 now precedes a_2 in the left-to-right scanning order.

Once the ALT is sorted, LINESCAN continues to process the ALT points as described above.

DEVELOPMENT AND SPECIALIZATION OF THE HSR ALGORITHM

In writing the HSR program, the basic logic of LINESCAN was implemented. Unlike LINESCAN, which was not expressly designed for real-time applications, HSR was written in assembly language. Some features implemented in LINESCAN were judged unnecessary for the visual simulator application. Chief among these was the "implicitly defined line" feature of LINESCAN. This feature allows polygons to intersect and project through one another. Without this feature, polygons projecting through one another subvert the scanning logic, producing incorrect and distracting images. In a driving simulator, intersecting polygonal objects usually represent car crashes; hence, these are events which should be distracting.

Some major changes to the basic logic of LINESCAN were implemented with the object of saving time. Recall that, when LINESCAN processes the ALT, as each point is encountered, a flag associated with that polygon is set to signify that the scan has "entered" that polygon. Then as each successive point is encountered, a search of the flags is used to determine which and how many flags are set. Performing even a short search at each point encountered on each scanline would consume a large fraction of the time allowed between frames in a real-time system. In the HSR algorithm, a table of polygon numbers is kept and updated as each new polygon is "entered" by the scan. The number of elements in the table is kept in a variable. Unless this variable indicates that the scan is "in" more than one polygon at a time, no "depth sort" is required and no search need be made for polygons flagged as "in." When a "depth sort" is required, the polygons which must be depth sorted are readily accessible by table reference.

Another change to the basic LINESCAN logic also involved sorting. LINESCAN sorts the ALT once for each scan. Recall that this step is required because the ALT is disordered when lines of different slopes intersect. Rather than sort the ALT for each scan, a simple test for ALT order was devised and performed at each point of the ALT. When disorder is found, the ALT is sorted. In simple scenes, this disorder occurs for about 8-10 of the possible 512 lines in a frame. Even very complex scenes require fewer than 20 ALT sorts. Hence, the savings in time are substantial.

REFERENCES

1 I E SUTHERLAND
A head-mounted three dimensional display
Proceedings of the Fall Joint Computer Conference Vol 33 Part I pp 757-764 1968

2 B ELSON
Color TV generated by computer to evaluate spaceborne systems
Aviation Week and Space Technology October 1967

3 *IDIOM-2—Interactive graphic display terminal*
The Computer Display Review Vol 5 pp 201-214 1972
GML Corporation Lexington Massachusetts

4 J BOUKNIGHT
An improved procedure for generation of half-tone computer graphics presentations
Communications of the ACM Vol 13 Number 9 pp 527-536 September 1970

Computer animation of a bicycle simulation

by JAMES P. LYNCH and R. DOUGLAS ROLAND

Cornell Aeronautical Laboratory, Inc.
Buffalo, New York

INTRODUCTION

For years, printed output was the only means of communication between the computer and man. This limitation dictated that only the technically skilled could interpret the reams of computer printout with its lists of numbers and specialized codes.

For certain types of computer usage, such as accounting, numbers may be the most meaningful form of output which can be presented to the user. Solutions to other problems, however, may represent functional relationships of intangible variables. In this case plots of output data provide a much faster means of communication between the computer and the human. There is a class of problems for which neither numerical nor plotted output provide sufficient reality for rapid user comprehension. One such area is the simulation of the dynamics of tangible physical systems such as airplanes, automobiles and bicycles. Fortunately, a means of communication is becoming practical which provides immediate visual interpretation of simulation results; not only for the analyst but for the layman as well. This mediumi s the computer animated graphics display.

The early development of computer animated graphics displays was spurred by several investigators. Bill Fetter of the Boeing Company created an animated human figure in 1960 and a carrier landing film in 1961.[1] Ed Zajac of Bell Telephone Laboratories produced a computer generated movie of a tumbling communications satellite in 1963.[2] Frank Sinden, also of Bell Laboratories, generated an educational computer animated film about gravitational forces acting on two bodies.[3] Two other investigators deserve mention, Ken Knowlton of Bell Labs for his computer animation language (BEFLIX)[4] and Ivan Sutherland for his interactive computer animation work.[5] Interested readers will find an excellent bibliography on the subject in Donald Weiner's survey paper on computer animation.[6]

In early 1971, Cornell Aeronautical Laboratory, Inc., (CAL), began a research program, sponsored by Schwinn Bicycle Company, devoted to the development of a comprehensive digital computer simulation of a bicycle and rider. This simulation would be used to study the effects of certain design parameters on bicycle stability and control. Phase II of this research effort included the development of a computer graphics display program which generates animated movies of the bicycle and rider maneuvers being simulated. It is this graphics display capability that is described herein.

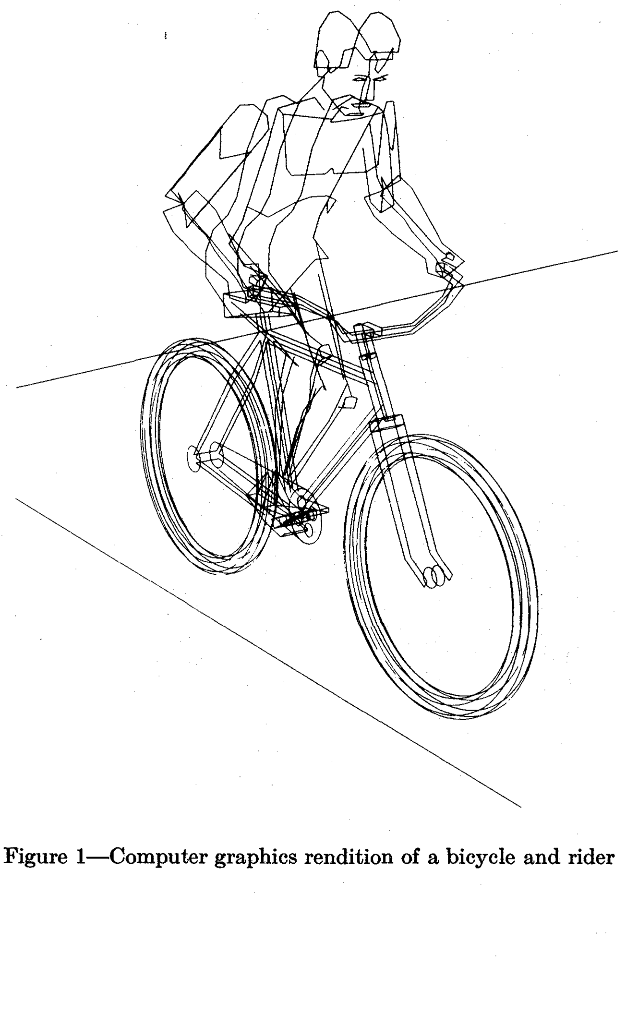


Figure 1—Computer graphics rendition of a bicycle and rider

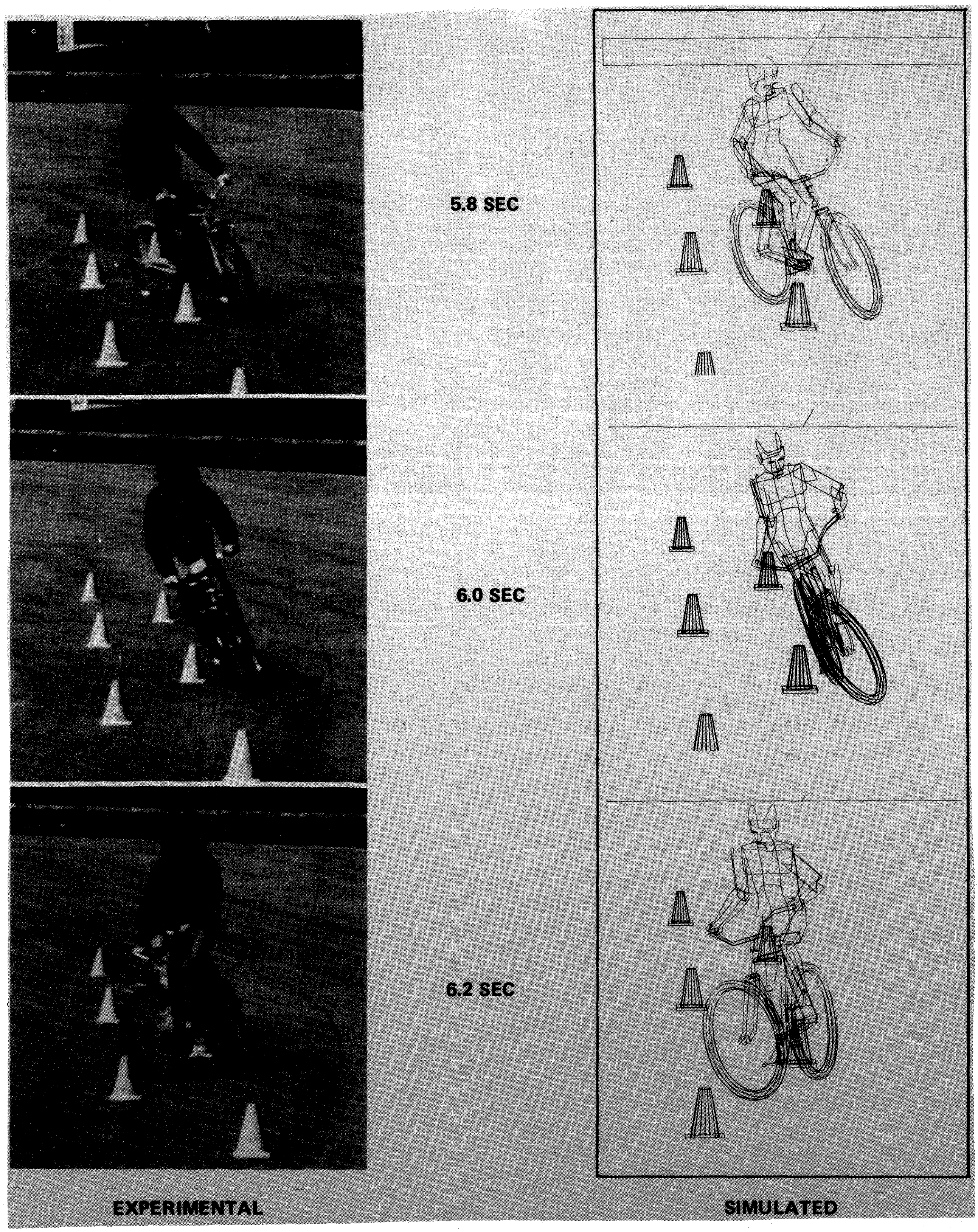


Figure 2—Bicycle slalom maneuver

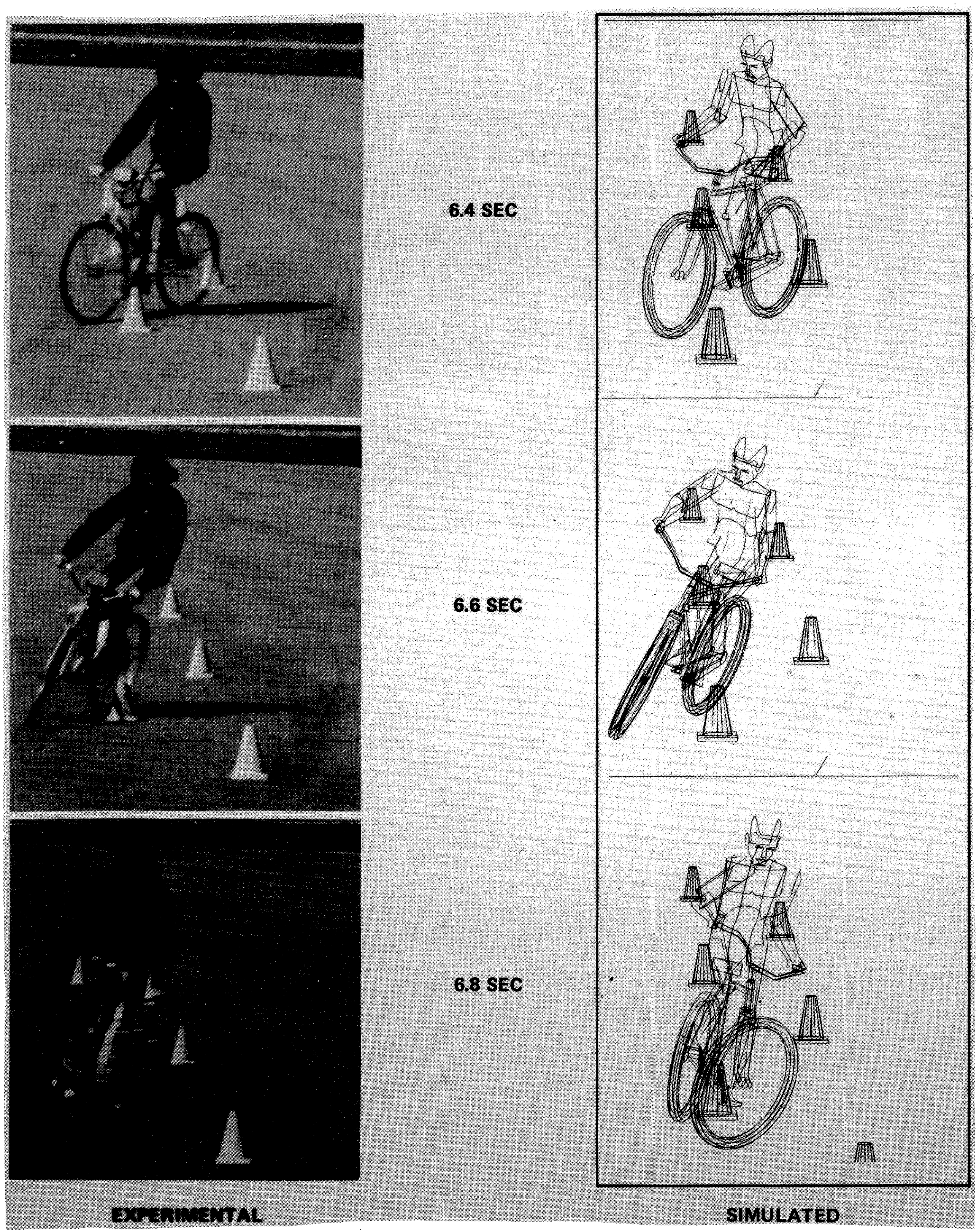


Figure 2 (Cont'd)

Computer Graphics activities at the Cornell Aeronautical Laboratory range from everyday use of general purpose plotting facilities by many programmers to highly complex computer-generated radar displays. One of the more fascinating computer graphics applications has been the Single Vehicle Accident Display Program, developed at CAL for the Bureau of Public Roads by C. M. Theiss.[9] This program converts automobile dynamics simulation data into a sequence of computer animated pictures used to generate motion picture film of the event. The demonstrated usefulness of this capability spurred the development of a graphics program for the Schwinn Bicycle Simulation.

BICYCLE GRAPHICS PROGRAM FEATURES

The Schwinn Bicycle Graphics Program provides a complete and flexible perspective graphics package capable of pictorially documenting the results of the bicycle simulation. The salient features of the graphics program are:

1. The program can plot a perspective picture of a bicycle and rider, positioned and oriented as per the simulation data.
2. The line drawing of the bicycle and rider can be easily changed to fit simulation or esthetic requirements.
3. The program can produce single pictures or animated movies.
4. Background objects, such as roadways, houses, obstacles, etc., can be plotted in the scene.
5. The "frame rate" for animated films can be adjusted for "slow motion" or normally timed action.
6. The program is written to simulate a 16 mm movie camera, so that "photographing" a scene is accomplished by specifying a set of standard camera parameters.
7. The program's "camera" can be set to automatically pan, zoom, remain fixed, or operate as on a moving base.
8. Any of the above characteristics may be changed during a run.

Figure 1 shows a typical frame from a bicycle simulation movie.

SIMULATION AND GRAPHICS SOFTWARE

Digital computer simulation of bicycle and rider

The computer simulation consists of a comprehensive analytical formulation of the dynamics of a bicycle-rider system stabilized and guided by a closed-loop rider control model. This computer simulation program will be used for bicycle design and development with particular consideration being given to the effects of various design parameters and rider ability on bicycle stability and maneuverability.

The bicycle-rider model is a system of three rigid masses with eight degrees of freedom: six rigid body degrees of freedom, a steer degree of freedom of the front wheel, and a rider lean degree of freedom. Included in the analysis are tire radial stiffness, tire side forces due to slip angle and inclination angle, the gyroscopic effects of the rotating wheels, as well as all inertial coupling terms between the rider, the front wheel and steering fork, and the rear wheel and frame.

Forty-four parameters of input data are required by the simulation program. These data include dimensions, weights, moments of inertia, tire side force coefficient, initial conditions, etc. The development of the simulation program has been supported by the measurement of the above physical characteristics of bicycles, the measurement of the side force characteristics of several types of bicycle tires and full scale experimental tests using an instrumented bicycle.

Solutions are obtained by the application of a modified Runge-Kutta step-by-step procedure to integrate equations of motion. Output is obtained from a separate output processor program which can produce time histories of as many as 36 variables (bicycle translational and angular positions, velocities, accelerations, and tire force components, etc.) in both printed and plotted format.

The simulation program, consisting of seven subroutines, uses approximately 170K bytes of core storage and requires about 4 seconds of CPU time per second of problem time when run on an IBM 370/165 computer. The output processor program uses approximately 200K bytes of core storage and requires about 5 seconds of CPU time per run. The total cost of both the simulation and output processor programs is approximately seven dollars per problem.

The mechanics of making a bicycle graphics movie

In addition to the printed and plotted output generated by the Schwinn Bicycle Simulation Program, a pecial "dynamics tape" is created for input to the bicycle graphics program. This dynamics tape contains, for each simulation solution interval, the bicycle's c.g. position (X, Y, Z coordinates), angular orientation (Euler angles), front wheel steer angle, and rider lean angle. All other pertinent information, such as the steering head caster angle, rider "hunch forward" angle, are fed to the graphics program via data cards, along with the stored three-dimensional line drawings of the bicycle and rider, and any desired backgrounds.

The bicycle graphics program searches the tape and finds the simulation time corresponding to the desired "frame time." Information is then extracted to draw the desired picture. The program mathematically combines the chassis, front fork and pedals to draw the bicycle, and mathematically combines the torso, left and right upper arms and forearms, and left and right thighs, calves and feet to draw the rider. Everything is so combined to yield a picture of a rider astride a bicycle assuming normal pedaling, leaning and handlebar grip. The correctly positioned three dimensional line drawings are transformed into a two dimensional picture plane, as specified by the program's camera parameters (location, orientation, focal length, etc.).

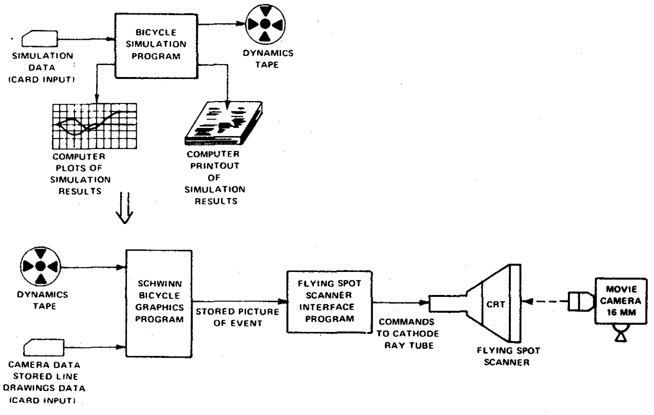

SIMULATION DATA (CARD INPUT)
BICYCLE SIMULATION PROGRAM
DYNAMICS TAPE
COMPUTER PLOTS OF SIMULATION RESULTS
COMPUTER PRINTOUT OF SIMULATION RESULTS
DYNAMICS TAPE
CAMERA DATA STORED LINE DRAWINGS DATA (CARD INPUT)
SCHWINN BICYCLE GRAPHICS PROGRAM
STORED PICTURE OF EVENT
FLYING SPOT SCANNER INTERFACE PROGRAM
COMMANDS TO CATHODE RAY TUBE
CRT
FLYING SPOT SCANNER
MOVIE CAMERA 16 MM

Figure 3—Steps in making Schwinn bicycle movie

An interface program converts the final line drawings into a set of commands to the CAL Flying Spot Scanner. The cathode ray tube beam of the Flying Spot Scanner traces out one frame of the movie while a 16 mm. movie camera records the image. Upon completion of the picture, the movie camera automatically advances one frame and the graphics program reads the next data (positions, angles, etc., of bicycle and rider) from the dynamics tape. The completed film will show animated motion, exactly as simulated by the computer, Figure 2. A block diagram of the movie making procedure is shown in Figure 3.

Bicycle motions displayed

For maximum realism and esthetic quality, seven distinct bicycle/rider motions were generated:

1. Bicycle chassis translation and rotation (6 degrees-of-freedom)

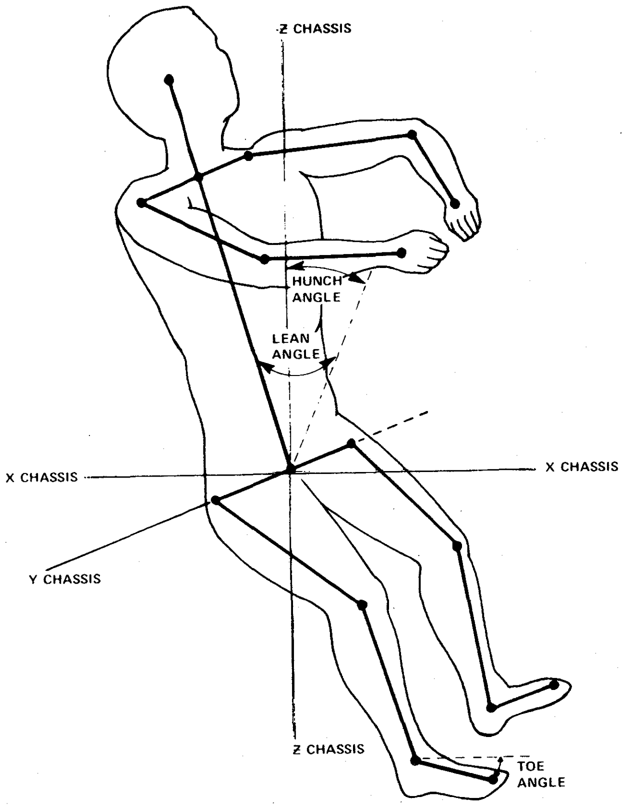


Figure 4—Joints used for rider display

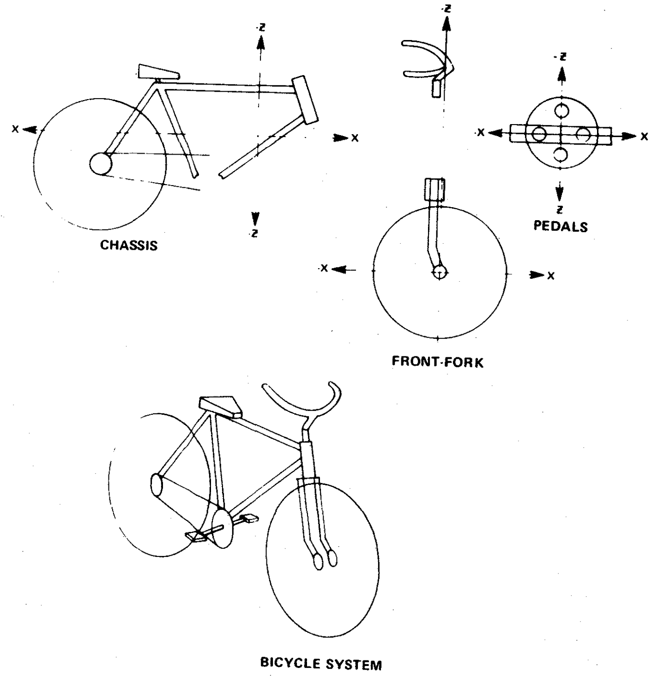


Figure 5—Sections used for bicycle display

2. Front wheel and handlebar steering
3. Bicycle crank and pedal rotation
4. Rider left-right leaning
5. Rider arm steering
6. Rider leg pedaling
7. Rider ankle flexing

Figure 4 shows the various body members and joints included in the rider. The separate parts of the bicycle are shown in Figure 5.

Modification of the basic graphics package

The Bureau of Public Roads graphics display program provided an excellent base from which to build the Schwinn Bicycle Graphics Program. A pre-stored line drawing, defined in its own coordinate system, is Euler transformed into fixed space and camera transformed into two dimensional picture space. Edge tests are performed to delete lines out of the field of view. Plotting any object (a line drawing) involves a call to the OBJECT subroutine

CALL OBJECT(TITLE, X, Y, Z, PHI, THETA, PSI)

Title refers to a particular stored line drawing, while X, Y, Z and PHI, THETA, PSI refer to the desired fixed space position and Euler angles at which the object is to be plotted. Subroutine OBJECT then does all the necessary transformations to plot the object. Plotting the chassis is straightforward, the chassis position and Euler angles are read directly from the dynamics tape.

Displaying the bicycle and rider

All segments of the bicycle and rider are displayed with the same mathematical approach. Parts are referenced by position and orientation to the chassis axis system, and this information is used to calculate the fixed space Euler angles and position. For example, the matrix equation relating points in the front fork axis system to corresponding points in fixed space is:

$$\begin{bmatrix} X_F \\ Y_F \\ Z_F \end{bmatrix} = [A] \left\{ [B] \begin{bmatrix} X_{STEER} \\ Y_{STEER} \\ Z_{STEER} \end{bmatrix} + \begin{bmatrix} X_{XF} \\ Y_{YF} \\ Z_{ZF} \end{bmatrix} \right\} + \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

where:

A is the standard Euler transformation matrix (chassis to fixed space)
B is the front-fork system to chassis axis transformation matrix
$(X_{STEER}, Y_{STEER}, Z_{STEER})$ are points in the front fork space
(X_{XF}, Y_{YF}, Z_{ZF}) is the front fork system connection point in the chassis system
(X, Y, Z) is the current fixed space position of the bicycle chassis
(X_F, Y_F, Z_F) is the front fork points specified in the fixed space set

The B matrix, of course, is a two rotational transformation, being a function of the caster angle and the steer angle. The Euler angles required by subroutine OBJECT can be determined by equating like terms of the standard Euler transformation with the overall transformation, [AB]=[A]*[B]
For instance:

$$\text{PHI} = \text{TAN}^{-1} \frac{\text{AB}(3,2)}{\text{AB}(3,3)}$$

$$\text{PSI} = \text{TAN}^{-1} \frac{\text{AB}(2,1)}{\text{AB}(1,1)}$$

$$\text{THETA} = \text{TAN}^{-1} \frac{-\text{AB}(3,1)*\text{SIN}(\text{PSI})}{\text{AB}(2,1)}$$

This procedure can be easily automated by a general subroutine which accepts the coefficients of the two transformation matrices and outputs the Euler angles.

Displaying the pedaling action

The pedal rotation angle is easily determined by tabulating the distance traveled by the chassis and relating it to the wheel size and gear ratio. The toe angle can be approximated by a cosine function of the pedal rotation angle.

$$\omega = \text{gear ratio}*\text{distance/wheelsize}$$

$$\text{Toe angle} = -.25*\cos(\omega)$$

An important simplifying assumption in the display of the leg pedaling motion is that the legs move up and down in a single plane. This makes trigonometric calculation of the joint locations straightforward and the object-to-chassis transformations simple one-rotation matrices. Once this information is determined, procedures similar to the front-fork manipulations are used. Three objects are required for each leg: the thigh, the calf, and the foot.

Displaying the torso

The torso must hunch forward (so that the arms may reach the handlebars) and lean to the left and right (real-world rider control action). The transformation between the torso axis system and the chassis system is determined by two rotations. This transformation is also used for determination of arm location in the chassis system.

Displaying the arms

Determination of the fixed space Euler angles of the arms is complicated by the fact that the elbow joint lies on a circular locus around the shoulder-to-handlebar line. Since the upper arm and forearm are assumed equal in length, the perpendicular distance from the elbow to the handlebar-to-shoulder line is known. A transformation matrix can be developed to convert points in the elbow circle plane to the torso system. A constant angle from the elbow circle plane's Y-axis defines a unique elbow point which can be transformed back into the chassis system. Once the elbow point is known, determination of the Euler angles of the arm is straightforward.

MOVIE PRODUCTION

Both the bicycle simulation program and the bicycle graphics program are run on CAL's IBM 370/165 computer. The flying spot scanner is interfaced with the central digital computer through an IBM 2909 asynchronous data channel. The flying spot scanner is a high resolution CRT display system used for plotting and scanning. The interface software provides all the controls required by the display to move the beam, advance the film, etc.

The Schwinn Bicycle Graphics program requires 250K bytes of core, and generally runs from 50¢ to 90¢ per frame in computing costs, depending on image complexity. No attempt at hidden line removal was planned for this phase.

FUTURE APPLICATIONS

The Schwinn Bicycle Graphics Program was designed as a research tool to demonstrate the capability of the bicycle simulation. Several computer animated movies have been produced of simulated bicycle maneuvers which compare well with full scale experimental maneuvers. At current production cost levels, only the most interesting runs are documented with the bicycle graphics program. The authors feel, however, that the advent of high speed intelligent computer terminals will allow the economical production of computer graphics. In the future the investigator will be able to view animated summaries of simulation results first, before referring to more detailed printed and plotted output data. The most gratifying result of this bicycle graphics capability is that the technically unskilled can share in the understanding that computer simulation is an emulation of reality, and has visible meaning in the everyday world.

ACKNOWLEDGMENT

The authors wish to express their gratitude to the Schwinn Bicycle Company for permission to present this work and also to Ronald B. Colgrove, CAL chief artist, for his excellent rendering of the bicycle rider used in the movie sequences.

REFERENCES

1 W A FETTER
Computer graphics in communication
McGraw-Hill New York 1965

2 E ZAJAC
Film animation by computer
New Scientist Vol 29 Feb 10, 1966 pp 346-349

3 F SINDEN
Synthetic cinematography
Perspective Vol 7 No 4 1965 pp 279-289

4 K KNOWLTON
A computer technique for producing animated movies
Joint Computer Conference AFIPS Conference Proceedings Vol 25 Baltimore Md Spartan 1964 pp 67-87

5 I SUTHERLAND
Perspective views that change in real time
Proceedings of 8th UAIDE Annual Meeting 1969 pp 299-310

6 D D WEINER
Computer animation—an exciting new tool for educators
IEEE Transactions on Education Vol E-14 No 4 Nov 1971

7 R D ROLAND JR D E MASSING
A digital computer simulation of bicycle dynamics
Cornell Aeronautical Laboratory Inc Technical Report No YA-3063-K-1 June 1971

8 R D ROLAND JR J P LYNCH
Bicycle dynamics, tire characteristics and rider modeling
Cornell Aeronautical Laboratory Inc Technical Report No YA-3063-K-2 March 1972

9 C M THEISS
Perspective picture output for automobile dynamics simulation
Prepared for Bureau of Public Roads by Cornell Aeronautical Laboratory Inc Technical Report No CPR-11-3988 January 1969

10 C M THEISS
Computer graphics displays of simulated automobile dynamics
Proceedings AFIPS Conference Spring 1969

An inverse computer graphics problem

by W. D. BERNHART

Wichita State University
Wichita, Kansas

The goal of a conventional computer perspective algorithm is to assist in the establishment of a scaled perspective view of a real or conceptual geometric object. The purpose of this paper is to present the required conditions for the inverse transformation; that is, given the perspective of an object, establish the required parameters used in generating the perspective and to a more restrictive extent, establish the original geometric definition of the object. Because this inverse mapping is from a two to three dimensional space, the method is approximate and is accomplished by the method of least squares based on certain *a priori* information regarding the geometrical object. The method does require a considerable amount of numerical computation, but is particularly well suited to a digital computer solution.

The need for this required transformation arose in the course of a problem associated with the determination of the coordinates of certain desired points which appeared in photographs of an event which occurred several years ago, wherein the desired points had been completely obliterated by recent construction activities. Thus, the first task was to establish the generating parameters for the photographs. The generating parameters are defined as six independent coordinates from which a photograph may be geometrically reproduced by considering a large number of points in the three-dimensional object space, and transforming these to the two-dimensional space of the photograph. These parameters consist of the coordinates of the point where the camera is located, the symmetric equations of the line along the optical axis of the camera, and a linear scale factor associated with the photograph, enlarged to any magnification. The treatment of a photograph as a true perspective is consistent with the paraxial ray tracing approximation of geometrical optics.

For the purpose of this analysis, all points will be defined in a rectangular Cartesian coordinate system as shown in Figure 1. The point where the camera is located is denoted by three independent coordinates, (X_e, Y_e, Z_e). In the context of traditional perspective terminology, this point is commonly described as the location of the eye or observer, and the point (X_o, Y_o, Z_o) is referred to as the center of interest of the object space or perspective center. A line through these two points is regarded as the optical axis of the camera and the plane perpendicular to this axis represents the picture plane, projection plane, or two-space photograph. The location of this plane in relation to the eye point requires the identification of a linear-scale factor which is associated with each photograph. The coordinates of

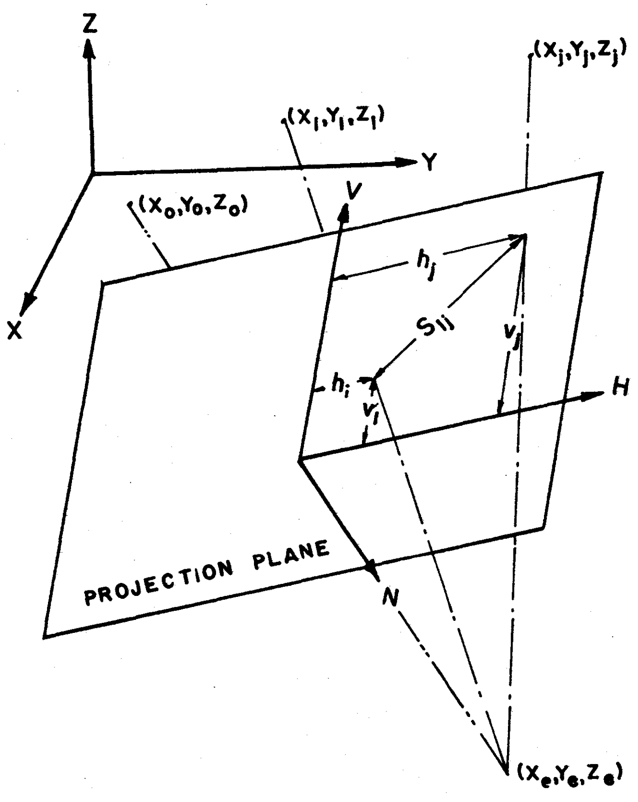


Figure 1—Projection plane and control points

the center of interest, (X_o, Y_o, Z_o), are not a unique set, as any point on the line passing through points 'e' and 'o' will require a particular value of the linear-scale factor to perspectively generate the object space into the projection plane space. For this analysis, the scale factor will be regarded as a constant and the six independent parameters, (X_e, Y_e, Z_e) and (X_o, Y_o, Z_o) will be determined such that the photograph may be geometrically reproduced in the perspective sense.

Before analyzing this particular problem, it will be necessary to present the required coordinate transformation that maps an arbitrary point 'i' in the object space to the projection-plane space. This perspective transformation has received considerable attention in computer graphics applications in the last decade.[1,2,3] A form which is particularly suited to the parameter identification problem is

$$n_i = R_o(1-\lambda) \tag{1}$$

$$h_i = \lambda(R_o^2/P_oD_i)\{-(X_i-X_o)(Y_e-Y_o) + (Y_i-Y_o)(X_e-X_o)\} \tag{2}$$

$$v_i = \lambda(R_o/P_oD_i)\{-[(X_i-X_o)(X_e-X_o) + (Y_i-Y_o)(Y_e-Y_o)](Z_e-Z_o)+(Z_i-Z_o)P_o^2\} \tag{3}$$

in which

$$P_o = [(X_e-X_o)^2+(Y_e-Y_o)^2]^{1/2}$$

$$R_o = [(X_e-X_o)^2+(Y_e-Y_o)^2+(Z_e-Z_o)^2]^{1/2} \tag{5}$$

$$D_i = R^2 - [(X_e-X_o)(X_i-X_o) + (Y_e-Y_o)(Y_i-Y_o)+(Z_e-Z_o)(Z_i-Z_o)] \tag{6}$$

and λ = the linear scale factor; $\lambda > 0$.

The coordinate normal to the picture plane is a constant and is of no particular interest other than as an aid in the estimation of a suitable photographic scale factor. For the case of a photograph, this normal coordinate is proportional to the focal length of a simple convergent camera lens. This particular form of the perspective mapping transformation is based on two-successive rotational transformations such that the plane defined by a line parallel to the Z-axis and the point 'e' also contains the V-axis of the projection plane. These two-successive rotations are defined as follows

$$\theta = \tan^{-1}[(Y_e+Y_o)/(X_e-X_o)] \tag{7a}$$

$$\beta = \sin^{-1}[(Z_e-Z_o)/R_o] \tag{7b}$$

A third rotation may be easily introduced by rotating the H, V-axes in the projection plane. It is important to note that distances measured in the projection plane would remain invariant with respect to this third rotation.

Returning to the original problem, the six desired parameters are determined by the method of least squares by considering four or more points in the object space whose rectangular coordinates are known or may be estimated with a high degree of accuracy. Next let $(S_{ij})_m$ denote the measured value of the distance between points i and j in the photograph. Thus, for 'n' such points, there are $m=n(n-1)/2$ corresponding measured distances. The calculated value of this associated distance in the projection plane is given by

$$(S_{ij})_c = [(h_j-h_i)^2+(v_j-v_i)^2]^{1/2} \tag{8}$$

and the six desired generating parameters are then obtained by expanding this calculated value in a multiple Taylor series, expressed as

$$\begin{aligned}(S_{ij})_c = (S_{ij})_a &+ \left(\frac{\partial S_{ij}}{\partial X_e}\right)_a \delta X_e + \left(\frac{\partial S_{ij}}{\partial Y_e}\right)_a \delta Y_e \\ &+ \left(\frac{\partial S_{ij}}{\partial Z_e}\right)_a \delta Z_e + \left(\frac{\partial S_{ij}}{\partial X_o}\right)_a \delta X_o \\ &+ \left(\frac{\partial S_{ij}}{\partial Y_o}\right)_a \delta Y_o + \left(\frac{\partial S_{ij}}{\partial Z_o}\right)_a \delta Z_o \\ &+ \text{higher-order terms}\end{aligned} \tag{9}$$

The subscript 'a' in Equation 9 denotes the evaluation for some assumed value of the six parameters. Thus, by neglecting the higher-order terms and minimizing the sum of the squares of the residuals between the calculated and measured values for the 'n' points

$$G = \sum_{k=1}^{m} [(S_{ij})_c - (S_{ij})_m]_k^2 \tag{10a}$$

and

$$\begin{aligned}&\frac{\partial G}{\partial X_e}=0, \quad \frac{\partial G}{\partial Y_e}=0, \quad \frac{\partial G}{\partial Z_e}=0, \\ &\frac{\partial G}{\partial X_o}=0, \quad \frac{\partial G}{\partial Y_o}=0, \quad \frac{\partial G}{\partial Z_o}=0\end{aligned} \tag{10b}$$

The six equations 10b, in general yield the six desired parameters after two to five iterations, depending on the initial assumed values of the parameters and the desired accuracy. Again, the scale factor is held constant throughout this iterative process. A different choice of λ will simply slide the coordinates of point 'o' along the line o-e without disturbing the iterated coordinates of point 'e'.

The writer has employed this procedure on several different controlled photographs with encouraging success.[4] These laboratory experiments yielded parameters estimates within 4 percent of their exact values. This error is largely attributed to the various unknowns associated with the optics of both the camera and enlarger, as both instruments were of commercial rather than laboratory quality. Recent experiments,[5] dealing with photogrammetric resectioning yielded considerably smaller errors. These experiments utilized a phototheodolite, spectroscopic flat quality glass plates and a mono comparator.

As mentioned earlier, the original need involved the determination of the coordinates of certain desired points which appeared in photographs of an event which occurred several years ago, wherein the desired points had been completely obliterated by construction activities. However, a sufficient number of points in the object space still existed such that the '*n*' required object-space coordinates described previously could still be easily obtained by field measurements. The desired points were located such that they appeared in two different photographs of the event. Thus, by iteratively determining the generating parameters for each photograph, the coordinates of the desired point were redetermined by solving for the intersection of the two lines associated with the point in each photograph.

REFERENCES

1 H R PUCKETT
Computer methods for perspective drawing
ARS-IAS Structures and Materials Conference Engineering Paper No 135 Palm Springs California April 1-3 1963

2 T E JOHNSON
Sketchpad III—A computer program for drawing in three dimensions
Proceedings Spring Joint Computer Conference 1963

3 W D BERNHART W A FETTER
Planar illustration method and apparatus
United States Patent Office No 3519997 July 7 1970

4 W D BERNHART
Determination of perspective generating parameters
ASCE Journal of the Surveying and Mapping Division Vol 94 No SU2 September 1968

5 L J FESSER
Computer-generated perspective plots for highway design evaluation
Federal Highway Administration Report No FHWA-RD-72-3 September 1971

Module connection analysis—A tool for scheduling software debugging activities

by FREDERICK M. HANEY

Xerox Corporation
El Segundo, California

INTRODUCTION

The largest challenge facing software engineers today is to find ways to deliver large systems on schedule. Past experience obviously indicates that this is not a well-understood problem. The development costs and schedules for many large systems have exceeded the most conservative, contingency-laden estimates that anyone dared to make. Why has this happened? There must be a plethora of explanations and excuses, but I think H. R. J. Grosch identified the common denominator in his article, "Why MAC, MIS and ABM will never fly."[1] Grosch's observation is essentially that for some large systems the problem to be solved and the system designed to solve it are in such constant flux that stability is never achieved. Even for some systems that are flying today, it is obvious that they came precariously close to this unstable, "critical mass" state.

It is my feeling that our most significant problem has been gross underestimation of the effort required to *change* (either for purposes of debugging or adding function) a large, complex system. Most existing systems spent several years in a state of gradual, painfully slow transition toward a releasable product. This transition was only partially anticipated and almost entirely unstructured; it was a time for putting out fires with little expectation about where the next one would occur.

The difficulties of stabilizing large systems are universal enough that our experience has resulted in several improved methods for estimating projects. Rules-of-thumb like "10 lines of code per man day" once sounded like extremely conservative allowance for the complexities of system integration and testing. J. D. Aron[2] has described a relatively elaborate technique for estimating total effort for large projects. Aron's technique is based on the estimated amount of code for a project and empirically observed distributions of various kinds of effort such as design, coding, module testing, etc. More recently Belady and Lehman described a mathematical model for the "meta-dynamics of systems in growth."[3] These schemes provide useful insights into the difficulties of designing and implementing large systems.

Even with these improved estimation techniques, however, we still face the threat of long periods of unstructured post-integration putting out of fires. We may know better how long this "final" debugging will take, but we are still at a loss to predict what resources will be required or what specific activities will take place. If we predict an 18 month period for "final testing," will management buy it? How can we peer into this hazy contingency portion of a schedule and predict in greater detail where bugs will occur, who will be needed to fix them, elapsed time between internal releases, etc.? Belady and Lehman suggest the need for a "micro-model" for system activities; i.e., a model based on internal, structural aspects of a system. This is essentially the objective of this paper. In the following sections, we will develop a very simple, but useful, technique for modeling the "stabilization" of a large system as a function of its internal structure.

The concrete result described in this paper is a simple matrix formula which serves as a useful *model* for the "rippling" effect of changes in a system. The real emphasis is on the use of the formula as a model; i.e., as an aid to understanding. The formula can certainly be used to obtain numeric estimates for specific systems, but its greater value is that it helps to *explain*, in terms of system structure and complexity, why the process of changing a system is generally more involved than our intuition leads us to believe.

The technique described here, called *Module Connection Analysis*, is based on the idea that every module pair (may be replaced by subsystem, component, or any other classification) of a system has a finite (possibly 0) probability that a change in one module will necessitate a change in any other module. By interpreting these probabilities and applying elementary matrix algebra, we can derive formulae for estimating the total number of "changes" required to stabilize a system and the staging of internal releases. The total number of changes, by module, is given by

$$A\times(I-P)^{-1},$$

where A is a row vector representing the initial changes per module, P is a matrix such that Pij is the probability that a change in module i necessitates a change in module j, and I is the $n\times n$ identity matrix. The number of changes required for each "internal release" is given by AP^k, $K=0, 1, \ldots,$ or by

$$A\times(I-P)^{-1}\times Uk, \qquad \begin{array}{l} k=1, 2, \ldots n, \\ Uk=(0, \ldots, \underset{\substack{\uparrow \\ k \text{ th element}}}{1}, \ldots 0) \end{array}$$

depending upon the release strategy. The derivations of these formulae are presented in the following section.

Module connection analysis is useful primarily as a tool for augmenting a designer's quantitative understanding of his problem. It produces quantitative estimates of the effects of module interconnections, an area in which intuitive judgment is generally inadequate.

THEORY OF MODULE CONNECTIONS

As a basis for our analysis, we postulate several characteristics of a system:

- A system is hierarchical in structure. It may consist of subsystems, which contain components, which contain modules or it may be completely general having n different levels of composition where an object at any level is composed of objects at the next lower level.
- At any level of the hierarchy, there may be some interdependence between any two parts of the system.
- If we view a system as a collection of modules (or, whatever object resides at the lowest hierarchical level), then the various interdependencies are manifested in terms of dependencies between all pairs of modules.

By dependence here, we mean that a change in one module may necessitate a change in the other. The fundamental axiom of module connection analysis is that intermodule connections are the essential culprit in elongated schedules. That a change in one module creates the necessity for changes in other modules, and these changes create others, and so on. Later, we will see that perfectly harmless-looking assumptions lead easily to sums like hundreds of changes required as a result of a single initial change. (The notions of hierarchy, interconnection, etc., used here are described at length in Reference 4.)

If we assume that a system consists of n "modules," then there are n^2 pairwise relationships of the form—

Pij = Probability that a change in module i necessitates a change in module j.

In the following, the letter "P" denotes the $n\times n$ matrix with elements pij. Furthermore, with each module i, there is associated a number Ai of changes that must be made in module i upon integration with the system. (Ai is approximately the number of bugs that show up in module i when it is integrated with the system.) If we let A denote a row vector with elements Ai, then we have the following:

A = total changes, by module, required at integration time, or at *internal release* 0.

AP = total changes required, by module, as a result of changes made in release 0, or total changes for *internal release* 1.

(Internal release $n+1$ is, roughly, a version of the system containing fixes for all first-order problems in internal release n.)

Now we observe that the i, jth element of P^2 is

$$\sum_{k=1}^{n} Pik\ Pkj,$$

which represents the sum of probabilities that a change in module i is propagated to module k and then to module j. Hence, the i, jth element of P^2 is the "two-step" probability that a change in module i propagates to module j. Or, AP^2 is the number of changes required in *internal release* 2.

The generalization is now obvious. The number of changes required in internal release k is given by AP^k and the total number of changes, T, is given by

$$T=A(I+P+P^2+P^3+\cdots).$$

Now we are interested to know whether or not the matrix power series in P converges; clearly, if it does not our system will never stabilize. To establish convergence of the power series, we appeal to matrix algebra (see Reference 5, for example) which tells us that the above series converges whenever the eigenvalues of P are less than 1 in absolute value. If this is the case, then the series converges and

$$T=A(I-P)^{-1}, \quad \text{where } I \text{ is the } n\times n \text{ identity matrix.}$$

We now have an extremely simple way to estimate the total number of changes required to stabilize a system as a linear function of a set of initial changes, A. Moreover, the number of changes at each release is given by the elements of AI, AP, AP^2, etc.

ESTIMATING TOTAL DEBUGGING EFFORT FOR A SYSTEM

The above theory suggests a simple procedure for estimating the total number of changes required to stabilize a system. The procedure is as follows:

(1) For each module pair, i, j, estimate the probability that a change in module i will force a change in module j. These estimates constitute the probability matrix P.
(2) From the vector A by estimating for each module i the number of "zero-order" changes, or changes required at integration time.
(3) Compute the total number of changes, by module:

$$T=A(I-P)^{-1}.$$

(4) Sum the elements of the column vector T to obtain the total number of changes, N.
(5) Make a simple extrapolation to "total time" based on past experience and knowledge of the environment. If past experience suggests a "fix" rate of d per week, then the total number of weeks required is N/d.

Hence if we have some estimate for the initial correctness (or "bugginess) of a system and for the intermodule connectivity (the probabilities), then we can easily obtain an estimate for the *total* number of changes that will be required to debug the system. The formula is a simple one in matrix notation, but the fact that we are dealing with matrices probably explains the failure of our intuition in understanding debugging problems.

In the following sections, we will show how the above formula can be used to aid our understanding of other aspects of the debugging process.

STAGING INTERNAL RELEASES

There are various strategies for tracking down bugs in a complex system. The most obvious are: (1) fix all bugs in one selected module and chase down *all* side effects, or, (2) fix all "first-order" bugs in each module, then fix all "second-order" bugs, and so on. The module connection model can aid in predicting release intervals for either approach.

For strategy (1) (one module at a time), the number of changes required to stabilize module i, given Ai initial changes, is given by

$$(p,\ldots,Ai,\ldots,0)(I-P)^{-1}$$

The product is a row vector with elements corresponding to the number of changes that must be made in each module as a result of the original changes. The total number of changes required to stabilize this one release is given by

$$Ai\sum_{k=1}^{n} Xik,$$

where the Xik are elements of $(I-P)^{-1}$. This strategy, then results in n internal releases where the time for release i is

$$Ai(\max_k Xik)\times(\text{time required per change})$$

and the total debug time after integration is

$$\sum_i (Ai \max_k Xik)\times(\text{time required per change})$$

With the second debugging strategy (make all "first-order" changes, then all "second-order" changes, etc.), the number of changes in the kth release is given by AP^k. That is, AP^k is a row vector with elements corresponding to the number of changes in each module for release k. The time required for release k is approximately

$$\max(AP^k)\times \text{time required per change.}$$

To determine the total number of releases for this strategy, we must examine A, AP, AP^2, until the number of changes AP^s in release s is small enough that the system is releasable. The total time for this strategy, then, is

$$\sum_{k=0}^{s} \max AP^k \times \text{time required per change.}$$

It is worth noting that both of the debug strategies described above evidence a "critical path" effect. The total time in each case is a sum of maximum times for each release. This effect corresponds to the well-known fact that debugging is generally a highly sequential process with only minor possibilities for making many fixes in parallel. This fact, coupled with the "amplification" of changes caused by rippling effects, certainly accounts for a large portion of many schedule slips.

REFINING THE INITIAL ESTIMATES

Module connection analysis is proposed as a tool for aiding designers and implementors. More than anything else, it is a rationale for making detailed quantitative estimates for what is generally called "contingency." Now, we must ask, "As a project progresses, how can we take advantage of actual experience to refine the initial estimates?" The module connection model is based on two objects: A, the vector of initial changes; and P, the matrix of connection probabilities between the modules. Both A and P can be revised simply as live data become available.

As each module i is integrated into the system, the number Ai of initial changes becomes apparent.

Using updated values for the vector A, we can recompute the expected total number of changes and the revised release strategy.

The elements, Pij, of the matrix P can be revised periodically if sufficient data is kept on changes, their causes, and their after-effects. One simple way to do this is to keep a record for each module as follows:

Module i ________

description of change	caused by which module?	other modules affected

After a relatively large sample of data is available, the above forms can be used to revise P as follows:

$$Pij = \frac{\text{number of changes in } j \text{ caused by } i}{\text{total changes made to } i}.$$

The revised matrix P can be used to revise earlier estimates for total effort and release strategies.

AN EXAMPLE OF MODULE CONNECTION ANALYSIS

The following example is based on the Xerox Universal Timesharing System. Eighteen actual subsystems are used as "modules." Estimates for connection probabilities and initial changes are made in the same way that they would be made for a new system, except that some experience and "feel" for the system were used to obtain realistic numbers. (Thanks to G. E. Bryan, Xerox Corporation, for helping to construct this example.)

The 18×18 probability connection matrix for this example is given in Figure 1. The matrix is relatively sparse; moreover, most of the nonzero elements have a value of .1. Most the larger elements lie on the diagonal

$$\begin{matrix}
.2 & .1 & 0 & 0 & 0 & .1 & 0 & .1 & 0 & .1 & .1 & .1 & 0 & 0 & 0 & .1 & 0 & 0 \\
0 & .2 & 0 & 0 & .1 & .1 & .1 & 0 & 0 & 0 & 0 & 0 & .1 & .1 & .1 & 0 & .1 & 0 \\
0 & 0 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & .1 & 0 & .2 & 0 & .1 & .1 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & .1 & 0 & .1 & 0 \\
.1 & 0 & 0 & 0 & .4 & .1 & .1 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .1 & 0 \\
.1 & 0 & 0 & 0 & 0 & .3 & .1 & 0 & 0 & .1 & 0 & 0 & 0 & .1 & 0 & 0 & .1 & 0 \\
.1 & 0 & 0 & .1 & .2 & .1 & .3 & .1 & 0 & .1 & 0 & 0 & 0 & .1 & 0 & .1 & .1 & 0 \\
.1 & .1 & 0 & .1 & .2 & 0 & .1 & .4 & 0 & .1 & 0 & 0 & 0 & .1 & 0 & 0 & 0 & .1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
.1 & 0 & 0 & 0 & 0 & .1 & .1 & .1 & 0 & .4 & .2 & .1 & .2 & .1 & .1 & .1 & .1 & .1 \\
.1 & 0 & 0 & .1 & 0 & 0 & 0 & 0 & 0 & .2 & .3 & .1 & 0 & 0 & 0 & 0 & 0 & 0 \\
.2 & 0 & 0 & 0 & 0 & .1 & 0 & 0 & 0 & 0 & .2 & .3 & 0 & 0 & .1 & .1 & 0 & 0 \\
.1 & .1 & 0 & 0 & 0 & .1 & .1 & .1 & 0 & .2 & .1 & 0 & .3 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .2 & 0 \\
0 & 0 & 0 & 0 & .1 & 0 & .1 & 0 & 0 & .1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .3
\end{matrix}$$

Figure 1—Probability connection matrix, P

INITIAL AND FINAL CHANGES

Module	Initial Changes	Total Required Changes
1	2	241.817
2	8	100.716
3	4	4.44444
4	6	98.1284
5	28	248.835
6	12	230.976
7	8	228.951
8	28	257.467
9	4	4.44444
10	8	318.754
11	40	238.609
12	12	131.311
13	16	128.318
14	12	157.108
15	12	96.1138
16	28	150.104
17	28	188.295
18	40	139.460
TOTALS	296	2963.85

Figure 2

corresponding to the fact that the subsystems are relatively large so that the probability of ripple within a subsystem is relatively large.

The total number of changes required in each module are given in Figure 2. It is interesting to note which modules require the most changes and to observe that six modules account for 50 percent of the changes.

Figure 3 illustrates the one-release-per-module debug strategy. That is, we repair one module and all side effects, then another module, and so on.

This strategy is rather erratic since the time between releases, which is determined by the maximum number of fixes in one module, ranges from 4 to 95 indiscriminately. If we adopt this strategy, we may want to select the worst module first and continue using the worst module at each step. We will see, however, that this strategy is far from optimal because it does not take maximum advantage of opportunities to make fixes in parallel.

ONE RELEASE PER MODULE

Release		Maximum Changes in One Module
1		4.41764
2		11.8619
3		4.44444
4		8.84029
5		67.8994
6		24.7185
7		20.3720
8		85.8099
9		4.44444
10		35.2976
11		95.2147
12		22.5608
13		39.7013
14		15.0000
15		15.0000
16		35.0000
17		35.0000
18		66.5554
"CRITICAL PATH"	TOTAL	592.138

Figure 3

A more effective release strategy is illustrated in Figure 4. This strategy assumes all first-order changes in release 1, all second order changes in release 2, etc. Figure 4 shows, for each release, the maximum number of changes in one module and the total number of changes. The reader who has worked on a large system will, no doubt, recognize the painfully slow convergence pattern. In this case, the system is assumed to be ready for external release when the "maximum changes per module" becomes less than one.

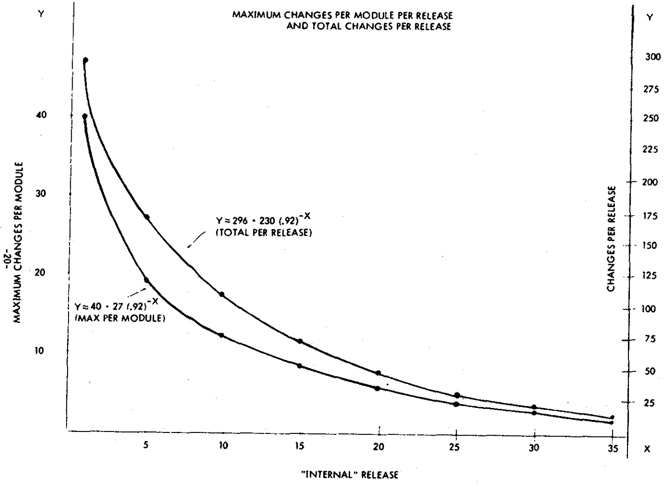


Figure 4—"Internal" release

If we assume the "critical path" changes are made at an average rate of about 1 per day, then Figure 4 is fairly representative of experience with the first release of UTS. The total number of changes on the "critical path" is 338, so that approximately 15 months would

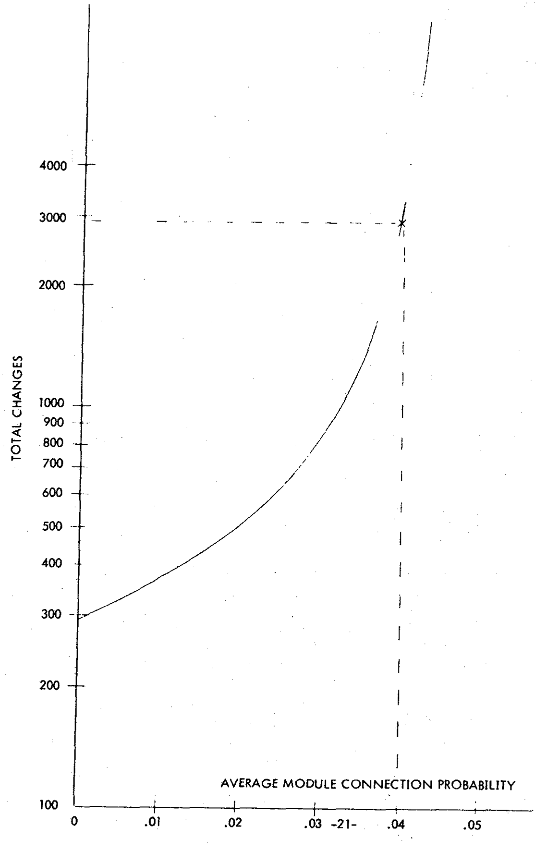


Figure 5—Total changes as a function of "average connection probability"

be required to stabilize the system for the first external release.

To conclude this example, let us take a brief look at the relationship between "total changes" and the probability of intermodule connection. The probabilities in the connection matrix above have an average value of approximately .04. What is the result if we assume the same relative distribution of probabilities in the matrix, but reduce the average by dividing each element by a constant?

Figure 5 shows the total number of changes as a function of "average probability of module connection" under the above assumption. This curve shows that our example is precariously close to "critical mass" and that any small improvement in the connection probabilities results in significant payoff.

OTHER APPLICATIONS OF MODULE CONNECTION ANALYSIS

The value of module connection analysis is its simplicity. The computations can be performed easily by a small (less than 50 lines) program written in APL, BASIC, or whatever language is available. Used on-line, the technique is useful for experimenting with various design approaches, implementation strategies, etc. Three examples of this use of the model are described below:

Estimating new work

If the designers, or managers, of a system have kept detailed records of the module-module changes in the system (as described above), then the matrix P is a reliable estimator of the "ripple factor" for the system. It can be used to predict, and stage, the effort to stabilize the system after any set of changes. If we postulate a major improvement release of the system, then we can assume, for example, that the new program code falls into two categories: (1) independent code particular to a new function and, (2) code that necessitates changes in an existing module. By estimating the number of changes, bi, to each module i, we can estimate the total number of changes to restabilize the system:

$$\text{Total changes} = (b_1, b_2, \ldots, b_n)(I-P)^{-1}.$$

The previously described computations can be used to estimate release intervals and total time for the improvement release.

To be more realistic, it may be useful in the above computation to use $bi+ei$ as the estimated changes in the module, where ei represents the number of changes required in module i by previous activity.

Evaluating design approaches

The best time to guarantee success of a system development effort is in the early design stages when architecture of the system is still variable. There is much to be gained by selecting an appropriate "decomposition" (see Reference 4). of the system into subsystems, components, etc. During this stage of a project, module connection analysis is a useful tool for evaluating various decompositions, interfacing techniques, etc. It is a simple, *quantitative* way of estimating the *modularity* of a system, the ever-present objective that no one knows exactly how to achieve. By fixing some of his assumptions about intermodule connections, a designer can experiment with various system organizations to determine which are the least likely to achieve "critical mass."

Evaluating implementation approaches

The reader who performs some simple experiments with the formulas described here is likely to be very surprised at the results. Even an extremely sparse connection matrix with very low probabilities can result [examine $(I-P)^{-1}$] in very large "ripple factors." It is also interesting to experiment with small perturbations in the connection matrix and observe the profound effect they can have on the "ripple factor." One becomes convinced more than ever before that it is necessary to minimize connections between modules, localize changes, and simplify the process of making changes.

The most impressive gains come from minimizing the probabilities of intermodule propagation of changes. A reduction of the average probability by as little as 5 or 10 percent can cause a significant reduction in the "ripple factor." Additional improvement can result from improvements in techniques for making changes. The total debug time is essentially linear with respect to the time required to make a change, but the multiplier (total number of changes) can be so large that any reduction in the time-per-change results in enormous savings.

Module connection techniques are extremely useful in estimating the value of various implementation techniques and strategies. How are the module connection probabilities changed if we use a high-level implementation language? How much easier will it be to make changes? How much will we save, if any, by doing elaborate environment simulation and testing of each module before it is integrated with the system? Module connection analysis is a valuable augmentation of intuition in these areas and can be useful for generating cost justifications for approaches that result in significant savings.

CONCLUSION

The objective of this paper has been to describe a simple model for the effect of "rippling changes" in a large system. The model can be used to estimate the number of changes and a release strategy for stabilizing a system given any set of initial changes. The model can be criticized for being simplistic, yet it seems to describe the *essence* of the problem of stabilizing a system. It is clear, to the author at least, that experimentation with the module connection model could have prevented a significant portion of the schedule delay that occurred for many large systems.

REFERENCES

1 H R J GROSCH
Why MAC, MIS, and ABM won't fly
Datamation 17 Nov 1 1971 pp 71-72

2 J D ARON
Estimating resources for large programming systems
Software Engineering Techniques J N Buxton and B Randell (eds) April 1970

3 L A BELADY M M LEHMAN
Programming system dynamics or the meta-dynamics of systems in maintenance and growth
Research Report IBM Thomas J Watson Research Center Yorktown Heights New York July 1971

4 C ALEXANDER
Notes on the synthesis of form
Cambridge Mass Harvard University Press 1964

5 M MARCUS
Basic theorems in matrix theory
National Bureau of Standards Applied Mathematics Series #57 U S Government Printing Office January 1960

Evaluating the effectiveness of software verification—Pratical experience with an automated tool

by J. R. BROWN and R. H. HOFFMAN

TRW Systems Group
Redondo Beach, California

INTRODUCTION

From the point of view of the user, a reliable computer program is one which performs satisfactorily according to the computer program's specifications. The ability to determine if a computer program does indeed satisfy its specifications is most often based upon accumulated experience in using the software. This is due in part to general agreement that the quality of computer software increases as the software is extensively used and failures are discovered and corrected. In keeping with this philosphy, increasing emphasis has been placed on exhaustive testing of computer programs as the principal means of assuring sufficient quality.

Nevertheless, a significant problem which pervades all software development is a lack of knowledge as to *how much testing* of a software system or component constitutes sufficient verification. The major impact of this problem (if not adequately addressed) is evidenced by high cost of testing (as much as 50 percent of total project cost) and insufficient visibility of test effectiveness. As a result, we often lack sufficient confidence that the software will continue to operate successfully for unanticipated combinations of data in a real-world environment.

In recognition of the high cost and uncertainty of software verification, TRW Systems' Product Assurance Office initiated a company-funded effort to improve upon current testing methodology. Much of the effort has been directed toward development of some general purpose automated software "tools" which would provide significant aid in performance of a software quality assurance activity. The desirable extent to which the "general purpose" and "automated" characteristics should be pursued has received considerable study, as did a precise definition of "significant aid." The result of the study, experimentation, design and development thus far conducted comprises the TRW **P**roduct **A**ssurance **C**onfidence **E**valuator (PACE) system, an evolving collection of automated tools which provide support in various phases of software testing.

Examination of a typical software testing process results in identification of four fundamental activities: test planning, production, execution and evaluation. Examination of the overall cost and schedule impact resulting from manual performance of these activities reveals the reasons for many testing efforts being less complete and successful than expected. With emphsais upon those tasks which are often neglected due to the menial aspect of their performance, PACE development was planned to complement manual testing efforts with automated utilities. Early planning and study efforts indicated a need to give emphasis to the ability of the system to meet diverse (and probably changing) user needs. To adequately cope with this requirement a number of events (instances) were identified at which operational releases of interim PACE capability would be most beneficial. Practical applications of the capabilities produced by each PACE instance would then provide meaningful direction for subsequent releases.

The initial PACE instance was the FLOW program to support test evaluation activities. FLOW monitors statement usage during test execution, thus providing a basic evaluation of test effectiveness. The results produced by FLOW, in particular the statement usage frequencies, are similar to the program profiles discussed by Knuth in Reference 1. In addition, FLOW supports the test planning activity by indicating the unexercised code and, consequently, the additional tests required for more comprehensive testing.

FLOW PROGRAM DESCRIPTION

Purpose

During the software development process, a question frequently asked (and seldom if ever answered satis-

```
  PSN                              STATEMENT
 0        ELEMENT SPEAR
 0              PROGRAM SPEAR(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
 0              DIMENSION A(10),B(10),R(20)
 0              NAMELIST/TESTIN/N,NR,A,B
 1          100 READ(5,TESTIN)
 2              CALL SRANK(A,B,R,N,RS,T,NDF,NR)
 3              WRITE(6,6001)A
                  :
 0        ELEMENT SRANK
 0              SUBROUTINE SRANK(A,B,R,N,RS,T,NDF,NR)
 0              DIMENSION A(1),B(1),R(1)
 1              FNNN=F(N)
 2              IF(NR-1)5,10,5
                  :
15              KT=1
16              CALL TIE(R,N,KT,TSA)
17              CALL TIE(R(N+1),N,KT,TSB)
18              FKTN=F(KT)+F(N)
19              IF(TSA)60,55,60
                  :
 0        ELEMENT TIE
 0              SUBROUTINE TIE(R,N,KT,T)
 0              DIMENSION R(1)
 1              T=0.0
 2              Y=0.0
 3            5 X=1.0E38
 4              IND=0
 5              DO 30 I=1,N
 6              IF(R(I)-Y)30,30,10
 7           10 IF(R(I)-X)20,30,30
 8           20 X=R(I)
 9              IND=IND+1
10           30 CONTINUE
11              IF(IND)90,90,40
12           40 Y=X
13              CT=0.0
14              DO 60 I=1,N
15,16           IF (R(I) EQ.X) CT=CT+1.0
17           60 CONTINUE
18,19           IF (CT.NE.0) IF(KT-1) 75,80,75
20              GO TO 5
21           75 T=T+CT*(CT-1.0)/2.0
22              GO TO 5
23           80 CONTINUE
24              ICT=CT
25              T=T+F(ICT)/12.0
26              GO TO 5
27           90 RETURN
 0              END
```

Figure 1—Sample program with pseudo statement numbers

factorily) is: "How much testing is enough?" There appears to be vital interest in the subject,[2,3,4] but too little in the way of practical applications has been accomplished in the past to provide any final answers. We feel strongly that a measure of the variety of ways in which a computer program is tested (or not tested) can combine to form a software "experience index", and quantification of the index supports evaluation of both the computer program and testing thoroughness. Based on this premise, the FLOW program was developed to: (1) support assessment of the extensiveness with which a computer program is tested, (2) provide a variety of quantified indices summarizing program operation, and, (3) support efforts to create a more comprehensive but less costly test process. The objective of FLOW is not to find errors, per se, but to quantitatively assess how thoroughly a program has been tested and to support test planning by indicating the portions of code which are not exercised by existing test cases.

Method

FLOW analyzes the source code of a computer program and instruments the code in a manner which permits subsequent compilation and makes possible monitored execution of the program. This technique is representative of one of several approaches toward software measurement technology described by Kolence.[5] A complete application of FLOW provides for an accumulation of frequencies with which selected program elements (e.g., statements, small segments of code, subprograms, etc.) are exercised as the program is being tested. There are optional levels of detail at which usage

```
**QAFLOW MAP PRINT**
-----------------------
   ELEMENT SPEAR      CUMULATIVE TIME       .0780 SECONDS
PSEUDO NOS.   FREQ   PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ
      1 TO  7=     1
-----------------------
   ELEMENT RANK       CUMULATIVE TIME      .6430 SECONDS
PSEUDO NOS.   FREQ   PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ
      1 TO  7=    20         8 TO  9=  200        10 TO 11=    90        12 TO 13=    20
     14 TO 14=   200        15 TO 17=   20        18 TO 22=     0        23 TO 23=    20
     24 TO 24=     2
-----------------------
   ELEMENT SRANK       CUMULATIVE TIME      .0860 SECONDS
PSEUDO NOS.   FREQ   PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ
      1 TO  5=     1         6 TO 10=    0        11 TO 11=     1        12 TO 14=    10
     15 TO 22=     1        23 TO 26=    0        27 TO 29=     1        30 TO 30=     0
     31 TO 32=     1
-----------------------
   ELEMENT TIE      CUMULATIVE TIME     1.2350 SECONDS
PSEUDO NOS.   FREQ   PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ    PSEUDO NOS.   FREQ
      1 TO  2=     2         3 TO  4=   22         5 TO  6=   220         7 TO  7=   110
      8 TO  9=    65        10 TO 10=  220        11 TO 11=    22        12 TO 13=    20
     14 TO 15=   200        16 TO 16=   20        17 TO 17=   200        18 TO 19=    20
     20 TO 22=     0        23 TO 26=   20        20 TO 27=     2
**QAFLOW USAGE SUMMARY AFTER    1589 NUMBER PAIRS(ENTRY/EXIT SEGMENTS)
  THE SUBJECT PROGRAM CONTAINS    90 EXECUTABLE STATEMENTS.
  THE TEST DATA EXERCISED    72 OF THESE STATEMENTS.
  THE TEST EFFECTIVENESS RATIO AT THE STATEMENT LEVEL IS    .80
  THE PROGRAM CONTAINS 1 TERMINATION POINTS, ONLY ONE OF WHICH WAS
EXECUTED. THE CORRECTED TEST EFFECTIVENESS RATIO IS    .80
THE PROGRAM CONTAINS    4 ENTRY POINTS. THE TEST DATA EXERCISED    4.
THE TEST EFFECTIVENESS RATIO AT THE ENTRY POINT LEVEL IS    1.00
```

Figure 2—FLOW execution frequency summary

monitoring can be performed. The desired level is selected by the user and controlled by input.

Typical use of the complete FLOW capability involves the application of three distinct FLOW elements. The first of these is QAMOD, the code analysis and instrumentation module. The QAMOD module sequentially analyzes each statement of a FORTRAN source program and accomplishes the following:

1. The first executable statement of each element (i.e., sub-routine or main program) is assigned a pseudo statement number (PSN) of one. Each subsequent statement (assuming that the most detailed monitoring is opted) is assigned a sequential PSN and the statements are displayed with their assigned number as illustrated in Figure 1.* Statements are referenced by element name and PSN during subsequent FLOW processing.
2. The code is instrumented by the insertion of transfers to the FLOW execution monitor subprogram, QAFLOW. The function of the transfers is the generation of a recording file containing the sequence of statements exercised duing test execution.

Upon completion of the analysis and instrumentation of the source program, the instrumented version of the program is output to a file for subsequent compilation and execution. QAFLOW is appended to the program prior to execution with test data.

The third FLOW module, QAPROC, provides summary statistics on the frequency of use of program elements as well as detailed trace information and an indication of the effectiveness of the test. QAPROC accesses the statement execution recording file generated by execution of the instrumented subject program and produces an evaluation and summary of the test case executed. The recording file is sequentially accessed and the data are assimilated into an internal table. At times designated by the input control options, a display is printed (Figure 2) which includes the following:

1. A map, delineated by subroutine, indicating the number of executions which have been recorded for each statement.
2. Statistics indicating the percentage of the total executable statements which were exercised at least once.
3. Statistics indicating the percentage of the total number of subroutines which were executed at least once.
4. A list of the names of subroutines which were not executed.
5. Total execution time spent in each subroutine.

Frequencies derived by FLOW from a number of separate tests of the subject program may be combined to provide a cumulative measure of the comprehensiveness of all testing applied to the program.

At the option of the user, detailed trace information can be displayed. The trace depicts the sequence in which statements (referenced by pseudo statement number) were exercised during program execution. A complete trace display for one test of the SPEAR program is illustrated in Figure 3. In addition, time of entry to each subroutine is recorded and displayed to support timing studies.

The information in Figure 3 is interpreted as follows:

- Execution is initiated at pseudo statement number (PSN)1 of the main program SPEAR at time 2.474;
- Subroutine SRANK is called from PSN 2 of SPEAR at time 2.479;
- Subroutine RANK is called following the sequential execution of PSN 1, 2 and 3 of SRANK;
- Upon entry to subroutine RANK, PSN 1 and 2 are executed 10 times before proceeding to PSN 3;
- When execution of RANK reaches PSN 24, control is returned to subroutine SRANK at PSN 4.

The value of the FLOW trace information in understanding an otherwise complex logic structure can be appreciated by following the execution of subroutine TIE (using the program listing in Figure 1).

The interaction of the three FLOW modules is illustrated in Figure 4 with a description of inputs and outputs for a typical application.

CASE STUDIES

In early planning for the capability which FLOW should provide, consideration was given to the requirements of the various phases of the software testing process. Because of the resulting flexibility of the FLOW program, successful use has been reported from a number of diverse applications. Major usage has been in two areas: (1) assessment of testing effectiveness, and

* The program shown is a modification of the Spearman Rank Correlation Coefficient program from the IBM Scientific Subroutine Package.[6] Figure 1 shows a portion of the main program SPEAR and the subroutine SRANK (lines omitted indicated by ⋮). The complete subroutine TIE is included to support later reference in this report.

```
**QAFLOW TRACE PRINT**
  ELEMENT SPEAR   TIME =    2.4740
 1- 2,
  ELEMENT SRANK   TIME =    2.4790
 1- 3,
  ELEMENT RANK    TIME =    2.4850
 1- 2     (10 TIMES)
 3- 9,     12- 14,     8- 11,     14- 14,     8- 11
14- 14,     8- 11,    14- 14,      8- 11,    14- 14
                  :
14- 14,     8-  9,    14- 14,      8-  9,    14- 14
 8-  9,    14- 14,     8-  9,     12- 17,    23- 24
  ELEMENT SRANK   TIME =    2.8040
 4- 4,
  ELEMENT RANK    TIME =    2.8100
   1- 2  (10 TIMES)
 3-  9,    12- 14,     8-  9,     14- 14,     8-  9
                  :
  ELEMENT SRANK   TIME =    3.7410
17- 17,
  ELEMENT TIE     TIME =    3.7470
 1- 10,     5-  7,    10- 10,      5-  7,    10- 10
 5-  7,    10- 10,     5-  7,     10- 10,     5-  7
10- 10,     5-  7,    10- 10,      5-  7,    10- 10
 5-  7,    10- 10,     5-  7,     10- 17,    14- 15
17- 17,    14- 15,    17- 17,     14- 15,    17- 17
14- 15,    17- 17,    14- 15,     17- 17,    14- 15
17- 17,    14- 15,    17- 17,     14- 15,    17- 17
14- 15,    17- 19,    23- 26,      3-  6,    10- 10
 5- 10,     5-  7,    10- 10,      5-  7,    10- 10
 5-  7,    10- 10,     5-  7,     10- 10,     5-  7
                  :
10- 15,    17- 17,    14- 15,     17- 17,    14- 15
17- 17,    14- 15,    17- 17,     14- 17,    14- 15
17- 17,    14- 15,    17- 17,     14- 15,    17- 17
14- 15,    17- 17,    14- 15,     17- 19,    23- 26
 3-  6,    10- 10,     5-  6,     10- 10,     5-  6
10- 10,     5-  6,    10- 10,      5-  6,    10- 10
                  :
17- 17,    14- 15,    17- 17,     14- 15,    17- 19
23- 26,     3-  6,    10- 10,      5-  6,    10- 10
 5-  6,    10- 10,     5-  6,     10- 10,     5-  6
                  :
10- 10,     5-  6,    10- 10,      5-  6,    10- 11
27- 27,
  ELEMENT SRANK   TIME =    4.3700
18- 22,    27- 29,    31- 32,
  ELEMENT SPEAR   TIME =    4.4270
 3- 7,
```

Figure 3—FLOW execution trace display

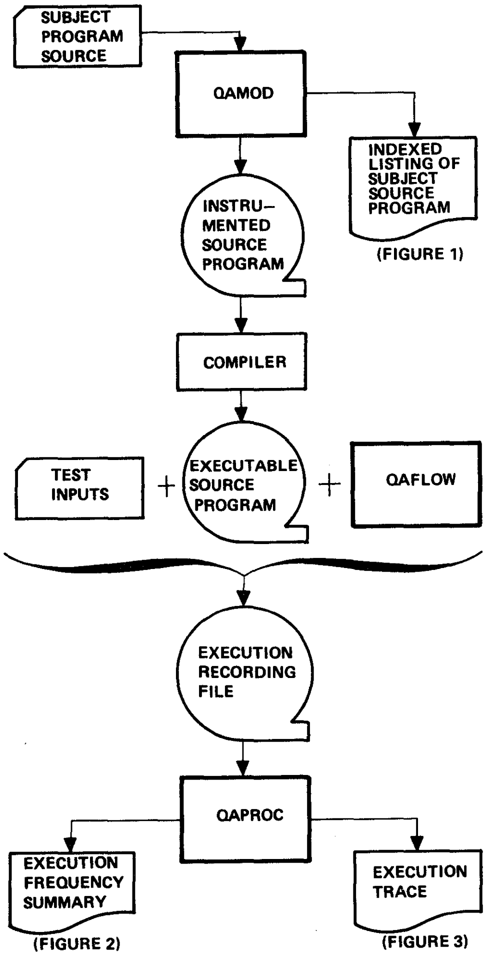


Figure 4—FLOW program overview

(2) analysis and solution of software problems difficult to solve with conventional techniques. Brief descriptions of several such applications are included here and grouped accordingly.

Test effectiveness

- Houston Operations Predictor/Estimator (HOPE)

The HOPE program is used by NASA/MSC for orbit determination and error analyses on the Apollo Missions. It contains approximately 500 subprograms including 80,000 lines of code. Over a four year period of program development, cases had been added to the test data file as required until the file consisted of 33 separate cases which required 4.5 hours of computer time and 35-50 man-hours of test results validation. Although developers were aware that redundant testing was being performed, it was impractical to delete any of the cases from the file. Because of the criticality of the program's accuracy, removal of any test case without precise proof of its impact on verification effectiveness could not be allowed. In addition, the tight schedule of the project did not permit detailed manual appraisal of each test case.

The FLOW program provided the means of determining the areas of HOPE which were tested by each case. The first FLOW analysis disclosed that the 33 cases tested 85 percent of the subprograms and that one-half of this number were exercised by almost every case. Consideration of these statistics prompted the funding of extended analyses to produce a more effective test file. An incremental test planning activity was performed and a file of six cases was generated. These six cases tested 93 percent of the subprograms, but they required less than three hours of computer time and less than 24 man-hours of test results examination. Since the FLOW analyses indicate the areas of the program exercised by each case, these six cases can be selectively used at each update to assure maximum cost effectiveness.

- Navigation Simulation Processor (NAVPRO)

NAVPRO is the program used by NASA/MSC to process data from Apollo Command Module and Lunar Module onboard computer navigation simulation programs. NAVPRO contains approximately 75 subprograms and 4000 executable statements.

FLOW was applied to NAVPRO to assist in the generation of a comprehensive set of test cases. The first step was selection of a basic test from the cases which were then being used for verification. FLOW analysis of the effectiveness of this first test had surprising results; although the case exercised 45 percent of the NAVPRO code, it was apparent that the time span being simulated (and consequently the case execution time) could be reduced by 85 percent without significantly reducing the effectiveness of the test. By eliminating this redundant testing and then manually extending the input data with the goal of improving its effectiveness, the case was modified such that it tested 80 percent of the code in one-fourth the time required by the original case. By continued application of FLOW, a complete test file consisting of four cases was compiled which tested 98 percent of the executable statements. The 2 percent not tested were areas of the program dedicated to error terminations not considered worthy of verification at each program update. These were verified initially and will be retested only if modifications are made which specifically affect their operation.

- Skylab Activities Scheduling Program

The FLOW program was used by NASA/MSC to measure the comprehensiveness of a set of 20 test cases for 52 subroutines comprising a crew model for the Skylab Activities Scheduling Program. The testing which had been performed was thought to be adequate but, since the program is to be used for on-line mission support, documentary evidence of sufficient verification is especially important.

Each of the 20 test cases was executed and evaluated separately by FLOW, then the results were accumulated using one of the FLOW options. These cumulative results verified that the critical software for each of the subroutines was indeed exercised; thus, there was no requirement to apply FLOW in the modification or addition of test cases.

Although no direct manpower savings can be assigned to this application, the value of the confidence in the software and in the test cases due to the FLOW results is evident. The users also acknowledged the value of the trace capability of FLOW, since they easily diagnosed a program error which had been previously undiscovered in their testing.

- Program Anatomy Tables Generator (TABGEN)

TABGEN is a utility program developed for NASA/MSC as one of the components of the Automated Verification System (AVS). The functions of TABGEN are to perform syntax analysis of FORTRAN programs, segment the code into blocks of statements and generate tables describing each of these blocks (e.g. variables referenced, transfer destinations) and the logical relationships between blocks. TABGEN consists of approximately 25 subroutines and 2000 executable statements.

Through FLOW application to TABGEN, test cases were devised to test 100 percent of the executable statements. The developers and users of TABGEN are convinced of the value of thorough testing, due to the fact that no errors have occurred since delivery of TABGEN in November 1971. The original version of the program was not altered until April 1972, when new requirements made modifications necessary.

- Minuteman Operational Targeting Program (MOTP)

MOTP is used by USAF/SAC to generate the targeting constants which must be supplied to the guidance computers aboard the Minuteman II missile system. The program contains approximately 160 subprograms which are extensively overlaid. Prior to each delivery of an updated version to SAC, extensive validation must be performed. Because of the criticality of this validation exercise, a means of accurately measuring the testing effectiveness was clearly required.

To determine the applicability of FLOW to the MOTP verification effort, a particularly complex portion of the program was instrumented and then monitored during a complete targeting run. FLOW provided new information about portions of the program which were assumed to be exercised but, in fact, were not. The results of this application clearly demonstrated the value of using FLOW to complement verification efforts. The decision was made to incorporate FLOW as a standard testing procedure for future deliveries. Recommendations were also made for selective use of the FLOW logic trace feature to gain a clearer understanding of the more complex portions of the MOTP.

Problem solving

- Apollo Reference Mission Program (ARM)

The ARM program is used by NASA/MSC during Apollo missions for simulation of all activities (powered and free flight) from earth launch to re-entry. Because of its extensive use during Apollo and anticipated future applications, it is imperative that the program execution time be optimal; expecially in the areas of the program which receive the most use.

The FLOW program was applied to ARM to determine the most-used portions of the program during a typical mission simulation and to obtain execution time analyses.* Although the application did not produce any surprising results, the predictions of the developers were verified (i.e., timing had been of prime consideration during development). Careful examination of critical statements (those exercised more than 10,000 times during the run) resulted in some minor modifications to improve timing which, if extrapolated over their anticipated period of use, will result in noticeable cost savings.

- DRUM SLAB II

The DRUM SLAB II aerodynamic analysis program was developed for NASA/MSC to simulate the molecule impact force and direction on spacecraft surfaces

* Similar applications have been produced by Knuth using the FORDAP program.[1]

during re-entry. During checkout, the program always aborted after seven minutes of execution with an illegal operation apparently resulting from erroneous storage of data due to the complex computation of various indices. Several unsuccessful attempts were made to manually diagnose and solve the problem. Although the incorrect data storage was thought to be occurring throughout the run, it did not cause an abort until the density of the molecules began to increase rapidly at lower altitudes. It was not obvious which of the indices were being miscalculated or at what point they were computed. Because of the complex modelling of the program and the fact that the original developers were not available, the problem caused the development project to be discontinued after three months of unsuccessful debug efforts using conventional methods.

Several months later, after attending a FLOW demonstration, the manager in charge of the DRUM SLAB development requested that FLOW be applied in an attempt to diagnose the problem. By selective instrumentation of the DRUM SLAB program and application of the FLOW data trace option, the problem was found to originate at some point during execution of the first 800 lines of the main program. Then, by close examination of the statement execution trace for these 800 lines, the precise point at which the erroneous index computation occurred was determined. Three separate errors were found in the computation of various indices. Correction of these computations eliminated the store error and resulted in an apparently error-free execution until the run was terminated by the operator at 15 minutes (the maximum execution time specified for the run). Although limited funds and lack of personnel familiar with the DRUM SLAB program prohibited a complete verification of the modified program, the utility of FLOW was proven by the fact that the problem had been solved in 50 man-hours by personnel totally unfamiliar with the DRUM SLAB program.

- Minuteman Geometric Identification Data Program (GIDATA)

The Minuteman Geometric Identification Data (GIDATA) program has been used to generate absolute and relative radar data for tracking sites. Recently, the program was extensively modified to generate special purpose output. The FLOW program was applied to the GIDATA program before modification was started in hope that the analysis would give a better understanding of the program and, hence, aid in modification design. Some of the most useful information obtained from FLOW was:

- Subroutine level trace and usage summary
- Inefficient subroutine structure and calling sequences
- Areas where code was never used
- Relative subroutine timing indicating inefficient code

Using this information a better understanding of GIDATA was achieved and it became relatively simple to determine necessary modifications for reducing program execution time and core requirements. Upon completion of the GIDATA modifications, additional applications of FLOW will ensure comprehensive testing of the program.

- Navigation Simulation Processor (NAVPRO)

In generation of a particular test case for NAVPRO (program described in the previous sections of this report), a problem developed when the error flag indicating vehicle impact with the lunar surface was being set during execution. Since the flag was in global COMMON and could have been set in any of several subroutines during the integration, it was difficult to determine precisely where the error was occurring.

Since NAVPRO had already been instrumented for statement execution monitoring, the origin of the error was easily detected. By using a special option of FLOW, the value stored in the error flag location was checked at execution of each transfer during the run. The FLOW display disclosed the exact statement at which the vehicle impact flag was set and described the program logic flow immediately preceding the impact. The error, which was in the NAVPRO input data, was found and corrected.

- Earth Re-entry Orbit Determination Program (REPOD)

REPOD is a large multi-link program developed and used in support of Minuteman trajectory analysis and orbit determination. Since REPOD is an amalgamation of several older programs, the detailed flow through each of its 9 links is particularly difficult to understand. The trajectory link is one of the more complex and was therefore chosen for FLOW analysis to identify possible program improvements. The analysis of the trajectory link was particularly desirable because:

(1) A significant portion of the total REPOD execution time is spent in this link.
(2) It was felt, by the user, that the FLOW analysis would lead to significant improvement in program efficiency.

One application of FLOW provided some striking results in identification of blocks of statements which were exercised with unexpected high frequency. FLOW also: (1) identified portions of REPOD not used for selected input options, (2) displayed subroutine and statement trace data for given options, and (3) indicated primary areas of concern for subsequent program improvements.

Using the FLOW results as a guide, a detailed examination of the trajectory integration algorithm was initiated. The complete task culminated in significant reductions in execution time (for example, processing time for one function was cut from 67 seconds to 11 seconds) and optimum selection of error criterion and integration step size for improved program performance.

SUMMARY

The initial PACE instance described here responded to an important need in supporting assurance of comprehensively tested and more reliable software products. Although execution of all statements is by no means a conclusive measure of test effectiveness, it is considered an important first step in the improvement of conventional testing methodology.

Subsequent instances of PACE have produced:

- A program which displays unexercised statements and performs an analysis of the FORTRAN code to determine the conditions necessary for their execution;[3] the computation and input of significant parameters is highlighted to support test redesign activities.
- A program to determine all possible logical transfers and extrapolate these to construct and display all logic paths within a FORTRAN module.[7]
- A program to monitor the execution of transfers during program execution;[8] a test effectiveness ratio is calculated based upon actual versus potential transfers exercised (used either as an alternative or in conjunction with FLOW statement usage analyses).

Parallel research and development activities have resulted in:

- A FLOW-like program to produce statement usage frequency without the execution trace feature;[9] although the results are not as detailed as those produced by FLOW, the program operation is more efficient and therefore more easily applied to large systems.
- Well-defined steps for the adaptation of PACE technology to programming languages other than FORTRAN (e.g., assembly language, COBOL, JOVIAL).

This approach toward development of PACE technology has proved successful and has resulted in needed exposure and critique of concepts and techniques. PACE applications have already provided some very meaningful answers to a variety of participants (from programmer to procurer) in a number of software development activities. As was expected, each new application lends additional insight into the evaluation of existing PACE technology and provides vital information for direction of continued design and implementation.[10] Application of PACE capabilities has stimulated interest in the effectiveness of testing among TRW personnel and its customers and has provided a firm foundation upon which a long-neglected technology[2] can now advance.

ACKNOWLEDGMENTS

Without the cooperation of many individuals the collection and presentation of the FLOW usage results documented here would have been an extremely difficult task. Those particularly deserving of mention are A. C. Arterbery, K. W. Krause, Dr. E. C. Nelson, R' M. Poole, R. W. Smith and R. F. Webber.

REFERENCES

1 D E KNUTH
An empirical study of FORTRAN programs
Software-Practice and Experience Vol 1 pp 105-103 1971

2 F GRUENBERGER
Program testing and validating
Computing: A First Course 1968

3 J R BROWN et al
Automated software quality assurance: A case study of three systems
Presented at the ACM SIGPLAN Symposium Chapel Hill North Carolina June 21-23 1972

4 LTC F BUCKLEY
Verification of software programs
Computers and Automation February 1971

5 K W KOLENCE
A software view of measurement tools
Datamation January 1971

6 *System/360 scientific subroutine package (360A-CM-03X) version III programmer's manual*
IBM Application Program H20-0205-3

7 J R BROWN
Practical applications of automated software tools
To be published in the Proceedings of the Western Electronic Show and Convention (WESCON)

Los Angeles California September 19-22 1972

8 R W SMITH
Measurement of segment relationship execution frequency
TRW Systems (#72-4912.30-31) March 29 1972

9 R H HOFFMAN et al
Node determination and analysis program (NODAL) user's manual
TRW Systems (#18793-6147-RO-00) June 30 1972

10 J R BROWN R H HOFFMAN
Automating software development—A survey of techniques and automated tools
TRW Inc May 1972

A design methodology for reliable software systems*

by B. H. LISKOV**

The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

Any user of a computer system is aware that current systems are unreliable because of errors in their software components. While system designers and implementers recognize the need for reliable software, they have been unable to produce it. For example, operating systems such as OS/360 are released to the public with hundreds of errors still in them.[1]

A project is underway at the MITRE Corporation which is concerned with learning how to build reliable software systems. Because systems of any size can always be expected to be subject to changes in requirements, the project goal is to produce not only reliable software, but readable software which is relatively easy to modify and maintain. This paper describes a design methodology developed as part of that project.

Rationale

Before going on to describe the methodology, a few words are in order about why a design methodology approach to software reliability has been selected.† The unfortunate fact is that the standard approach to building systems, involving extensive debugging, has not proved successful in producing reliable software, and there is no reason to suppose it ever will. Although improvements in debugging techniques may lead to the detection of more errors, this does not imply that all errors will be found. There certainly is no guarantee of this implicit in debugging: as Dijkstra said, "Program testing can be used to show the presence of bugs, but never to show their absence."[3]

In order for testing to guarantee reliability, it is necessary to insure that all relevant test cases have been checked. This requires solving two problems:

(1) A complete (but minimal) set of relevant test cases must be identified.
(2) It must be possible to test all relevant test cases; this implies that the set of relevant test cases is small and that it is possible to generate every case.

The solutions to these problems do not lie in the domain of debugging, which has no control over the sources of the problems. Instead, since it is the system design which determines how many test cases there are and how easily they can be identified, the problems can be solved most effectively during the design process: The need for exhaustive testing must influence the design.

We believe that such a design methodology can be developed by borrowing from the work being done on proof of correctness of programs. While it is too difficult at present to give formal proofs of the correctness of large programs, it is possible to structure programs so that they are more amenable to proof techniques. The objective of the methodology presented in this paper is to produce such a program structure, which will lend itself to *informal* proofs of correctness. The proofs, in addition to building confidence in the correctness of the program, will help to identify the relevant test cases, which can then be exhaustively tested. When exhaustive testing is combined with informal proofs, it is reasonable to expect reliable software after testing is complete. This expectation is borne out by at least one experiment performed in the past.[4]

The scope of the paper

A key word in the discussion of software reliability is "complex"; it is only when dealing with complex sys-

* This work was supported by Air Force Contract No. F19(628)-71-C-0002.

** Present Address—Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts.

† The material in this section is covered in much greater detail in Liskov and Towster.[2]

tems that reliability becomes an acute problem. A twofold definition is offered for "complex." First, there are many system states in such a system, and it is difficult to organize the program logic to handle all states correctly. Second, the efforts of many individuals must be coordinated in order to build the system. A design methodology is concerned with providing techniques which enable designers to cope with the inherent logical complexity effectively. Coordination of the efforts of individuals is accomplished through management techniques.

The fact that this paper only discusses a design methodology should not be interpreted to imply that management techniques are unimportant. Both design methodology and management techniques are essential to the successful construction of reliable systems. It is customary to divide the construction of a software system into three stages: design, implementation, and testing. Design involves both making decisions about what precisely a system will do and then planning an overall structure for the software which enables it to perform its tasks. A "good" design is an essential first step toward a reliable system, but there is still a long way to go before the system actually exists. Only management techniques can insure that the system implementation fits into the structure established by the design and that exhaustive testing is carried out. The management techniques should not only have the form of requirements placed on personnel; the organization of personnel is also important. It is generally accepted that the organizational structure imposes a structure on the system being built.[5] Since we wish to have a system structure based on the design methodology, the organizational structure must be set up accordingly.*

CRITERIA FOR A GOOD DESIGN

The design methodology is presented in two parts. This section defines the criteria which a system design should satisfy. The next section presents guidelines intended to help a designer develop a design satisfying the criteria.

To reiterate, a complex system is one in which there are so many system states that it is difficult to understand how to organize the program logic so that all states will be handled correctly. The obvious technique to apply when confronting this type of situation is "divide and rule." This is an old idea in programming and is known as modularization. Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas:[7] "The connections between modules are the assumptions which the modules make about each other." Modules have connections in control via their entry and exit points; connections in data, explicitly via their arguments and values, and implicitly through data referenced by more than one module; and connections in the services which the modules provide for one another.

Traditionally, modularity was chosen as a technique for system production because it makes a large system more manageable. It permits efficient use of personnel, since programmers can implement and test different modules in parallel. Also, it permits a single function to be performed by a single module and implemented and tested just once, thus eliminating some duplication of effort and also standardizing the way such functions are performed.

The basic idea of modularity seems very good, but unfortunately it does not always work well in practice. The trouble is that the division of a system into modules may introduce additional complexity. The complexity comes from two sources: functional complexity and complexity in the connections between the modules. Examples of such complexity are:

(1) A module is made to do too many (related but different) functions, until its logic is completely obscured by the tests to distinguish among the different functions (functional complexity).
(2) A common function is not identified early enough, with the result that it is distributed among many different modules, thus obscuring the logic of each affected module (functional complexity).
(3) Modules interact on common data in unexpected ways (complexity in connections).

The point is that if modularity is viewed only as an aid to management, then any ad hoc modularization of the system is acceptable. However, the success of modularity depends directly on how well modules are chosen. We will accept modularization as the way of organizing the programming of complex software systems. A major part of this paper will be concerned with the question of how good modularity can be achieved, that is, how modules can be chosen so as to minimize the connections between them. First, however, it is necessary to give a definition of "good" modularity. To emphasize the requirement that modules be as disjoint as possible, and because the term "module" has been used so often and so diversely, we will discard it and define modularity as the division of the system into

* Management techniques intended to support the design methodology proposed in this paper are described by Liskov.[6]

"partitions." The definition of good modularity will be based on a synthesis of two techniques, each of which addresses a different aspect of the problem of constructing reliable software. The first, levels of abstraction, permits the development of a system design which copes with the inherent complexity of the system effectively. The second, structured programming, insures a clear and understandable representation of the design in the system software.

Levels of abstraction

Levels of abstraction were first defined by Dijkstra.[8] They provide a conceptual framework for achieving a clear and logical design for a system. The entire system is conceived as a hierarchy of levels, the lowest levels being those closest to the machine. Each level supports an important abstraction; for example, one level might support segments (named virtual memories), while another (higher) level could support files which consist of several segments connected together. An example of a file system design based entirely on a hierarchy of levels can be found in Madnick and Alsop.[9]

Each level of abstraction is composed of a group of related functions. One or more of these functions may be referenced (called) by functions belonging to other levels; these are the external functions. There may also be internal functions which are used only within the level to perform certain tasks common to all work being performed by the level and which cannot be referenced from other levels of abstraction.

Levels of abstraction, which will constitute the partitions of the system, are accompanied by rules governing some of the connections between them. There are two important rules governing levels of abstraction. The first concerns resources (I/O devices, data): each level has resources which it owns exclusively and which other levels are not permitted to access. The second involves the hierarchy: lower levels are not aware of the existence of higher levels and therefore may not refer to them in any way. Higher levels may appeal to the (external) functions of lower levels to perform tasks; they may also appeal to them to obtain information contained in the resources of the lower levels.*

Structured programming

Structured programming is a programming discipline which was introduced with reliability in mind.[11,12] Although of fairly recent origin, the term "structured programming" does not have a standard definition. We will use the following definition in this paper.

Structured programming is defined by two rules. The first rule states that structured programs are developed from the top down, in levels.* The highest level describes the flow of control among major functional *components* (major subsystems) of the system; *component names* are introduced to represent the components. The names are subsequently associated with code which describes the flow of control among still lower-level components, which are again represented by their component names. The process stops when no undefined names remain.

The second rule defines which control structures may be used in structured programs. Only the following control structures are permitted: concatenation, selection of the next statement based on the testing of a condition, and iteration. Connection of two statements by a *goto* is not permitted. The statements themselves may make use of the component names of lower-level components.

Structured programming and proofs of correctness

The goal of structured programming is to produce program structures which are amenable to proofs of correctness. The proof of a structured program is broken down into proofs of the correctness of each of the components. Before a component is coded, a specification exists explaining its input and output and the function which it is supposed to perform. (The specification is defined at the time the component name is introduced; it may even be part of the name.) When the component is coded, it is expressed in terms of specifications of lower level components. The theorem to be proved is that the code of the component matches its specifications; this proof will be given based on axioms stating that lower level components match their specifications.

The proof depends on the rule about control structures in two important ways. First, limiting a component to combinations of the three permissible control structures insures that control always returns from a component to the statement following the use of the

* In the Madnick and Alsop paper referenced earlier, the hierarchy of levels is strictly enforced in the sense that if the third level wishes to make use of the services of the first level, it must do so through the second level. This paper does not impose such a strict requirement; a high level may make use of a level several steps below it in the hierarchy without necessarily requiring the assistance of intermediate levels. The 'THE' system[8] and the Venus system[10] contain example of levels used in this way.

* The levels in a structured program are not (usually) levels of abstraction, because they do not obey the rule about ownership of resources.

component name (this would not be true if *goto* statements were permitted). This means that reasoning about the flow of control in the system may be limited to the flow of control as defined locally in the component being proved. Second, each permissible control structure is associated with a well-known rule of inference: concatenation with linear reasoning, iteration with induction, and conditional selection with case analysis. These rules of inference are the tools used to perform the proof (or understand the component).

Structured programming and system design

Structured programming is obviously applicable to system implementation. We do not believe that by itself it constitutes a sufficient basis for system design; rather we believe that system design should be based on identification of levels of abstraction.* Levels of abstraction provide the framework around which and within which structured programming can take place. Structured programming is compatible with levels of abstraction because it provides a comfortable environment in which to deal with abstractions. Each structured program component is written in terms of the names of lower-level components; these names, in effect, constitute a vocabulary of abstractions.

In addition, structured programs can replace flowcharts as a way of specifying what a program is supposed to do. Figure 1 shows a structured program for the top level of the parser in a bottom-up compiler for an operator precedence grammar, and Figure 2 is a flowchart containing approximately the same amount of detail. While it is slightly more difficult to write the structured program, there are compensating advantages. The structured program is part of the final program; no translation is necessary (with the attendant possibility of introduction of errors). In addition, a structured program is more rigorous than a flowchart. For one thing, it is written in a programming language and therefore the semantics are well defined. For another, a flowchart only describes the *flow of control* among parts of a system, but a structured program at a minimum must also define the data controlling its flow,

```
begin
integer relation;
boolean must_scan;
string symbol;
stack parse_stack;
must_scan := true;
push(parse_stack, eof_entry);
while not finished(parse_stack) do
  begin
  if must_scan then symbol := scan_next_symbol;
  relation := precedence_relation(top(parse_stack), symbol);
  perform_operation_based_on_relation(relation, parse_stack,
                                  symbol, must_scan)
  end
end
```

Figure 1—A structured program for an operator precedence parser

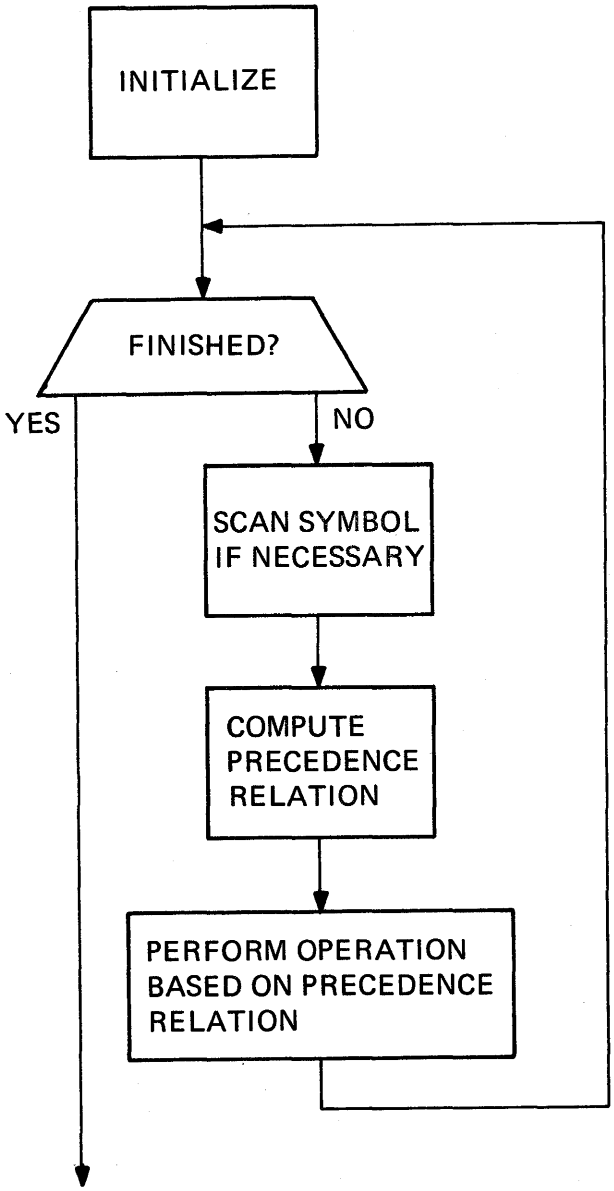

INITIALIZE
FINISHED?
YES
NO
SCAN SYMBOL IF NECESSARY
COMPUTE PRECEDENCE RELATION
PERFORM OPERATION BASED ON PRECEDENCE RELATION

Figure 2—Flowchart of an operator precedence parser

* A recent paper by Henderson and Snowden[13] describes an experiment in which structured programming was the only technique used to build a program. The program had an error in it which was the direct result of not identifying a level of abstraction.

so the description it provides is more concrete. In addition, it defines the arguments and values of a referenced component, and if a change in level of abstraction occurs at that point, then the data connection between the two components is completely defined by the structured program. This should help to avoid interface errors usually uncovered during system integration.

Basic definition

We now present a definition of good modularity supporting the goal of software reliability. The system is divided into a hierarchy of partitions, where each partition represents one level of abstraction, and consists of one or more functions which share common resources. At the same time, the entire system is expressed by a structured program which defines the way control passes among the partitions. The connections between the partitions are limited as follows:

(1) The connections in control are limited by the rules about the hierarchy of levels of abstraction and also follow the rules for structured programs.
(2) The connections in data between partitions are limited to the explicit arguments passed from the functions of one partition to the (external) functions of another partition. Implicit interaction on common data may only occur among functions within a partition.
(3) The combined activity of the functions in a partition support its abstraction and nothing more. This makes the partitions logically independent of one another. For example, a partition supporting the abstraction of files composed of many virtual memories should not contain any code supporting the existence of virtual memories.

A system design satisfying the above requirements is compatible with the goal of software reliability. Since the system structure is expressed as a structured program, it should be possible to prove that it satisfies the system specifications, assuming that the structured programs which will eventually support the functions of the levels of abstraction satisfy their specifications. In addition, it is reasonable to expect that exhaustive testing of all relevant test cases will be possible. Exhaustive testing of the whole system means that each partition must be exhaustively tested, and all combinations of partitions must be exhaustively tested. Exhaustive testing of a single partition involves both testing based on input parameters to the functions in the partition and testing based on intermediate values of state variables of the partition. When this testing is complete, it is no longer necessary to worry about the state variables because of requirement 2. Thus, the testing of combinations of partitions is limited to testing the input and output parameters of the external functions in the partitions. In addition, requirement 3 says that partitions are logically independent of one another; this means that it is not necessary when combining partitions to test *combinations* of the relevant test cases for each partition. Thus, the number of relevant test cases for two partitions equals the sum of the relevant test cases for each partition, not the product.

GUIDELINES FOR SYSTEM DESIGN

Now that we have a definition of good modularization, the next question is how a system modularization satisfying this definition can be achieved. The traditional technique for modularization is to analyze the execution-time flow of the system and organize the system structure around each major sequential task. This technique leads to a structure which has very simple connections in control, but the connections in data tend to be complex (for examples see Parnas[14] and Cohen[15]). The structure therefore violates requirement 2; it is likely to violate requirement 3 also since there is no reason (in general) to assume any correspondence between the sequential ordering of events and the independence of the events.

If the execution flow technique is discarded, however, we are left with almost nothing concrete to help us make decisions about how to organize the system structure. The guidelines presented here are intended to help rectify this situation. First are some guidelines about how to select abstractions; these guidelines tend to overlap, and when designing a system, the choice of a particular abstraction will probably be based on several of the guidelines. Next the question of how to proceed with the design is addressed. Finally, an example of the selection of a particular abstraction within the Venus system[10] is presented to illustrate the application of several of the principles; an understanding of Venus is not necessary for understanding the example.

Guidelines for selecting abstractions

Partitions are always introduced to support an abstraction or concept which the designer finds helpful in thinking about the system. Abstraction is a very valuable aid to ordering complexity. Abstractions are introduced in order to make what the system is doing clearer and more understandable; an abstraction is a conceptual simplification because it expresses what is being done without specifying how it is done. The purpose of this section is to discuss the types of abstractions which may be expected to be useful in designing a system.

Abstractions of resources

Every hardware resource available on the system will be represented by an abstraction having useful characteristics for the user or the system itself. The abstraction will be supported by a partition whose functions map the characteristics of the abstract resource into the characteristics of the real underlying resource or resources. This mapping may itself make use of several lower partitions, each supporting an abstraction useful in defining the functions of the original partition. It is likely that a strict hierarchy will be imposed on the group of partitions; that is, other parts of the system may only reference the functions in the original partition. In this case, we will refer to the lower partitions as "sub-partitions."

Two examples of abstract resources are given. In an interactive system, "abstract teletypes" with end-of-message and erasing conventions are to be expected. In a multiprogramming system, the abstraction of processes frees the rest of the system from concern about the true number of processors.

Abstract characteristics of data

In most systems the users are interested in the structure of data rather than (or in addition to) storage of data. The system can satisfy this interest by the inclusion of an abstraction supporting the chosen data structure; functions of the partition for that abstraction will map the structure into the way data is actually represented by the machine (again this may be accomplished by several sub-partitions). For example, in a file management system such an abstraction might be an indexed sequential access method. The system itself also benefits from abstract representation of data; for example, the scanner in a compiler permits the rest of the compiler to deal with symbols rather than with characters.

Simplification via limiting information

According to the third requirement for good modularization, the functions comprising a partition support only one abstraction and nothing more. Sometimes it is difficult to see that this restriction is being violated, or to recognize that the possibility for identification of another abstraction exists.

One technique for simplification is to limit the amount of information which the functions in the partition need to know (or even have access to). An example of such information is the complicated format in which data is stored for use by the functions in the partition (the data would be a resource of the partition). The functions require the information embedded in the data but need not know how it is derived from the data. This knowledge can be successfully hidden within a lower partition (possibly a sub-partition) whose functions will provide requested information when called; note that the data in question become a resource of the lower partition.

Simplification via generalization

Another technique for simplification is to recognize that a slight generalization of a function (or group of functions) will cause the functions to become generally useful. Then a separate partition can be created to contain the generalized function or functions. Separating such groups is a common technique in system implementation and is also useful for error avoidance, minimization of work, and standardization. The existence of such a group simplifies other partitions, which need only appeal to the functions of the lower partition rather than perform the tasks themselves. An example of a generalization is a function which will move a specified number of characters from one location to another, where both locations are also specified; this function is a generalization of a function in which one or more of the input parameters is assumed.

Sometimes an already existing partition contains functions supporting tasks very similar to some work which must be performed. When this is true, a new partition containing new versions of those functions may be created, provided that the new functions are not much more complex than the old ones.

System maintenance and modification

Producing a system which is easily modified and maintained is one of our primary goals. This goal can be aided by separating into independent partitions functions which are performing a task whose definition is likely to change in the future. For example, if a partition supports paging of data between core and some backup storage, it may be wise to isolate as an independent partition those functions which actually know what the backup storage device is (and the device becomes a resource of the new partition). Then if a new device is added to the system (or a current device is removed), only the functions in the lower partition will be affected; the higher partition will have been isolated from such changes by the requirement about data connections between partitions.

How to proceed with the design

Two phases of design are distinguished. The very first phase of the design (phase 1) will be concerned with defining precise system specifications and analyzing them with respect to the environment (hardware or software) in which the system will eventually exist. The result of this phase will be a number of abstractions which represent the eventual system behavior in a very general way. These abstractions imply the existence of partitions, but very little is known about the connections between the partitions, the flow of control among the partitions (although a general idea of the hierarchy of partitions will exist), or how the functions of the partitions will be coded. Every important external characteristic of the system should be present as an abstraction at this stage. Many of the abstractions have to do with the management of system resources; others have to do with services provided to the user.

The second phase of system design (phase 2) investigates the practicality of the abstractions proposed by phase 1 and establishes the data connections between the partitions and the flow of control among the partitions. This latter exercise establishes the placement of the various partitions in the hierarchy. The second phase occurs concurrently with the first; as abstractions are proposed, their utility and practicality are immediately investigated. For example, in an information retrieval system the question of whether a given search technique is efficient enough to satisfy system constraints must be investigated.

A partition has been adequately investigated when its connections with the rest of the system are known and when the designers are confident that they understand exactly what its effect on the system will be. Varying depths of analysis will be necessary to achieve this confidence. It may be necessary to analyze how the functions of the partition could be implemented, involving phase 1 analysis as new abstractions are postulated requiring lower partitions or sub-partitions. Possible results of a phase 2 investigation are that an abstraction may be accepted with or without changes, or it may be rejected. If an abstraction is rejected, then another abstraction must be proposed (phase 1) and investigated (phase 2). The iteration between phase 1 and phase 2 continues until the design is complete.

Structured programming

It is not clear exactly how early structured programming of the system should begin. Obviously, whenever the urge is felt to draw a flowchart, a structured program should be written instead. Structured programs connecting all the partitions together will be expected by the end of the design phase. The best rule is probably to keep trying to write structured programs; failure will indicate that system abstractions are not yet sufficiently understood and perhaps this exercise will shed some light on where more effort is needed or where other abstractions are required.

When is the design finished?

The design will be considered finished when the following criteria are satisfied:

(1) All major abstractions have been identified and partitions defined for them; the system resources have been distributed among the partitions and their positions in the hierarchy established.

(2) The system exists as a structured program, showing how the flow of control passes among the partitions. The structured program consists of several components, but no component is likely to be completely defined; rather each component is likely to use the names of lower-level components which are not yet defined. The interfaces between the partitions have been defined, and the relevant test cases for each partition have been identified.

(3) Sufficient information is available so that a skeleton of a user's guide to the system could be written. Many details of the guide would be filled in later, but new sections should not be needed.*

An example from Venus

The following example from the Venus system[10] is presented because it illustrates many of the points made about selection, implementation, and use of abstractions and partitions. The concept to be discussed is that of external segment name, referred to as ESN from now on.

The concept of ESN was introduced as an abstraction primarily for the benefit of users of the system. The important point is that a segment (named virtual memory) exists both conceptually (as a place where a

* This requirement helps to insure that the design fulfills the system specifications. In fact, if there is a customer for whom the system is being developed, a preliminary user's guide derived from the system design could be a means for reviewing and accepting the design.

programmer thinks of information as being stored) and in reality (the encoding of that information in the computer). The reality of a segment is supported by an internal segment name (ISN) which is not very convenient for a programmer to use or remember. Therefore, the symbolic ESN was introduced.

As soon as the concept of ESN was imagined, the existence of a partition supporting this concept was implied. This partition owned a nebulous data resource, a dictionary, which contained information about the mappings between ESNs and ISNs. The formatting of this data was hidden information as far as the rest of the system was concerned. In fact, decisions about the dictionary format and about the algorithms used to search a dictionary could safely be delayed until much later in the design process. A collective name, the dictionary functions, was given to the functions in this partition.

Now phase 2 analysis commenced. It was necessary to define the interface presented by the partition to the rest of the system. Obvious items of interest are ESNs and ISNs; the format of ISNs was already determined by the computer architecture, but it was necessary to decide about the format of ESNs. The most general format would be a count of the number of characters in the ESN followed by the ESN itself; for efficiency, however, a fixed format of six characters was selected.

At this point a generalization of the concept of ESN occurred, because it was recognized that a two-part ESN would be more useful than a single symbolic ESN. The first part of the ESN is the symbolic name of the dictionary which should be used to make the mapping; the second part is the symbolic name to be looked up in the dictionary. This concept was supported by the existence of a dictionary containing the names of all dictionaries. A format had to be chosen for telling dictionary functions which dictionary to use; for reasons of efficiency, the ISN of the dictionary was chosen (thus avoiding repeated conversions of dictionary ESN into dictionary ISN).

When phase 2 analysis was over, we had the identification of a partition; we knew what type of function belonged in this partition, what sort of interface it presented to the rest of the system, and what information was kept in dictionaries. As the system design proceeded, new dictionary functions were specified as needed. Two generalizations were realized later. The first was to add extra information to the dictionary; this was information which the system wanted on a segment basis, and the dictionaries were a handy place to store it. The second was to make use of dictionary functions as a general mapping device; for example, dictionaries are used to hold information about the mapping of record names into tape locations, permitting simplification of a higher partition.

In reality, as soon as dictionaries and dictionary functions were conceived, a core of dictionary functions was implemented and tested. This is a common situation in building systems and did not cause any difficulty in this case. For one thing, extra space was purposely left in dictionary entries because we suspected we might want extra information there later although we did not then know what it was. The search algorithm selected was straight serial search; the search was embedded in two internal dictionary functions (a sub-partition) so that the format of the dictionaries might be changed and the search algorithm redefined with very little effect on the system or most of the dictionary functions. This follows the guideline of modifiability.

CONCLUSIONS

This paper has described a design methodology for the development of reliable software systems. The first part of the methodology is a definition of a "good" system modularization, in which the system is organized into a hierarchy of "partitions", each supporting an "abstraction" and having minimal connections with one another. The total system design, showing how control flows among the partitions, is expressed as a structured program, and thus the system structure is amenable to proof techniques.

The second part of the methodology addresses the question of how to achieve a system design having good modularity. The key to design is seen as the identification of "useful" abstractions which are introduced to help a designer think about the system; some methods of finding abstractions are suggested. Also included is a definition of the "end of design", at which time, in addition to having a system design with the desired structure, a preliminary user's guide to the system could be written as a way of checking that the system meets its specifications.

Although the methodology proposed in this paper is based on techniques which have contributed to the production of reliable software in the past, it is nevertheless largely intuitive, and may prove difficult to apply to real system design. The next step to be undertaken at MITRE is to test the methodology by conscientiously applying it, in conjunction with certain management techniques,[6] to the construction of a small, but complex, multi-user file management system. We hope that this exercise will lead to the refinement, extension and clarification of the methodology.

ACKNOWLEDGMENTS

The author wishes to thank J. A. Clapp and D. L. Parnas for many helpful criticisms.

REFERENCES

1 J N BUXTON B RANDELL (eds)
Software engineering techniques
Report on a Conference Sponsored by the NATO Science Committee Rome Italy p 20 1969

2 B H LISKOV E TOWSTER
The proof of correctness approach to reliable systems
The MITRE Corporation MTR 2073 Bedford Massachusetts 1971

3 E W DIJKSTRA
Structured programming
Software Engineering Techniques
Report on a Conference sponsored by the NATO Science Committee Rome Italy J N Buxton and B Randell (eds) pp 84-88 1969

4 F T BAKER
Chief programmer team management of production programming
IBM Syst J 11 1 pp 56-73 1972

5 M CONWAY
How do committees invent?
Datamation 14 4 pp 28-31 1968

6 B H LISKOV
Guidelines for the design and implementation of reliable software systems
The MITRE Corporation MTR 2345 Bedford Massachusetts 1972

7 D L PARNAS
Information distribution aspects of design methodology
Technical Report Department of Computer Science Carnegie-Mellon University 1971

8 E W DIJKSTRA
The structure of the "THE"—multiprogramming system
Comm ACM 11 5 pp 341-346 1968

9 S MADNICK J W ALSOP II
A modular approach to file system design
AFIPS Conference Proceedings 34 AFIPS Press Montvale New Jersey pp 1-13 1969

10 B H LISKOV
The design of the Venus operating system
Comm ACM 15 3 pp 144-149 1972

11 E W DIJKSTRA
Notes on structured programming
Technische Hogeschool Eindhoven The Netherlands 1969

12 H D MILLS
Structured programming in large systems
Debugging Techniques in Large Systems R Rustin (ed) Prentice Hall Inc Englewood Cliffs New Jersey pp 41-55

13 P HENDERSON R SNOWDEN
An experiment in structured programming
BIT 12 pp 38-53 1972

14 D L PARNAS
On the criteria to be used in decomposing systems into modules
Technical Report CMU-CS-71-101 Carnegie-Mellon University 1971

15 A COHEN
Modular programs: Defining the module
Datamation 18 1 pp 34-37 1972

A summary of progress toward proving program correctness

by T. A. LINDEN

National Security Agency
Ft. George G. Meade, Maryland

INTRODUCTION

Interest in proving the correctness of programs has grown explosively within the last two or three years. There are now over a hundred people pursuing research on this general topic; most of them are relative newcomers to the field. At least three reasons can be cited for this rapid growth:

(1) The inability to design and implement software systems which can be guaranteed correct is severely restricting computer applications in many important areas.
(2) Debugging and maintaining large computer programs is now well recognized as one of the most serious and costly problems facing the computer industry.
(3) A large number of mathematicians, especially logicians, are interested in applications where their talents can be used.

This paper summarizes recent progress in developing rigorous techniques for proving that programs satisfy formally defined specifications. Until recently proofs of correctness were limited to toy programs. They are still limited to small programs, but it is now conceivable to attempt to prove the correctness of small critical modules of a large program. This paper is designed to give a sufficient introduction to current research so that a software engineer can evaluate whether a proof of correctness might be applicable to some of his problems sometime in the future.

THE NATURE OF CORRECTNESS PROOFS

Given formal specifications for a program and given the text of a program in some formally defined language, it is then a well-defined mathematical question to ask whether the program text is correct with respect to those specifications. The mathematics necessary for this was originally worked out primarily by Floyd[1] and Manna.[2]

It must be made clear that a proof of correctness is radically different from the usual process of testing a program. Testing can and often does prove a program is incorrect, but no reasonable amount of testing can ever prove that a nontrivial program will be correct over all allowable inputs.

Example

The approach to proving programs correct which was developed and popularized by Floyd is still the basis for most current proofs of correctness. It is generally known as the method of inductive assertions. Let us begin with a simple example of the basic idea. Consider the flowchart in Figure 1 for exponentiation to a positive integral power by repeated multiplication. For simplicity, assume all values are integers. I have put assertions or specifications for correctness on the input and output of the program. We want to prove that if X and Y are inputs with $Y>0$, then the output Z will satisfy $Z=X^Y$. This assertion at the output is the specification for correctness of the program. The assertion at the input defines the input conditions (if any) for which the program is to produce output satisfying the output assertion. Note that the proof will use symbolic techniques to establish that the program is correct for all allowable inputs.

The proof technique works as follows: Somewhere within each loop we must add an assertion that adequately characterizes an invariant property of the loop. This has been done for the single loop flowchart of Figure 1. It is now possible to break this flowchart into tree-like sections such that each section begins and ends with assertions and no section contains a loop. This is shown in Figure 2 if one disregards the dashed-line boxes. We want to show that if execution of a section begins in a state with the assertion at its head true, then when the execution leaves that section, the assertion at the exit must also be true. By taking an assertion at the end of each of these sections and using the semantics of the program statement above it, one can generate an assertion which should have held before that statement if the assertions after it are to be guaranteed true. Working up the trees one then generates all the assertions in dashed-line boxes in Figure 2. Each section will then preserve truth from its first to its last assertions if the first assertion implies the assertion that was generated in the dashed-line box at the top. One thus gets the logical theorems or verification conditions given below each section. With a little thought it can now be seen that if these theorems can all be proven and if the program halts, then it will halt with the correct output values. In this case the theorems are obviously true. Halting can be proven by other techniques.

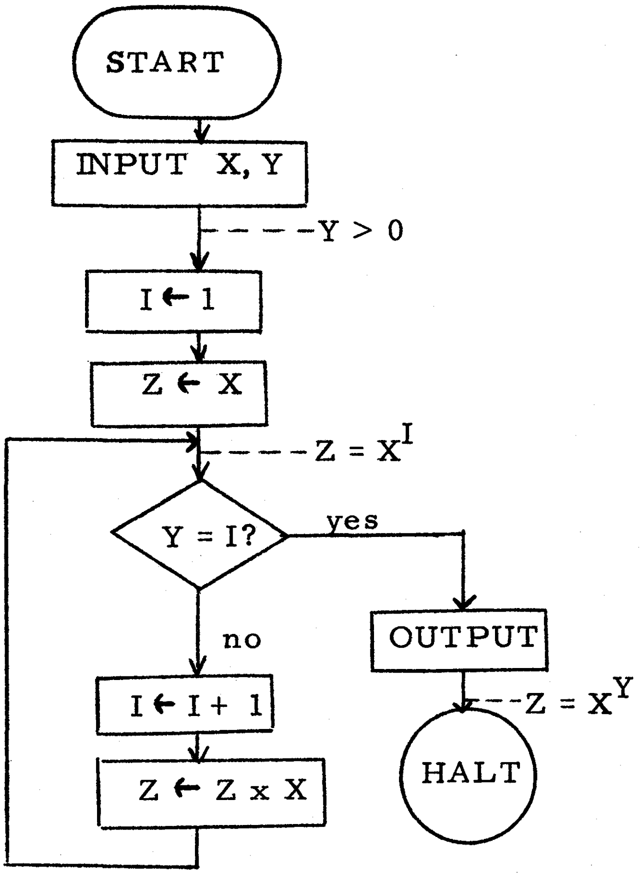


Figure 1—Exponentiation program

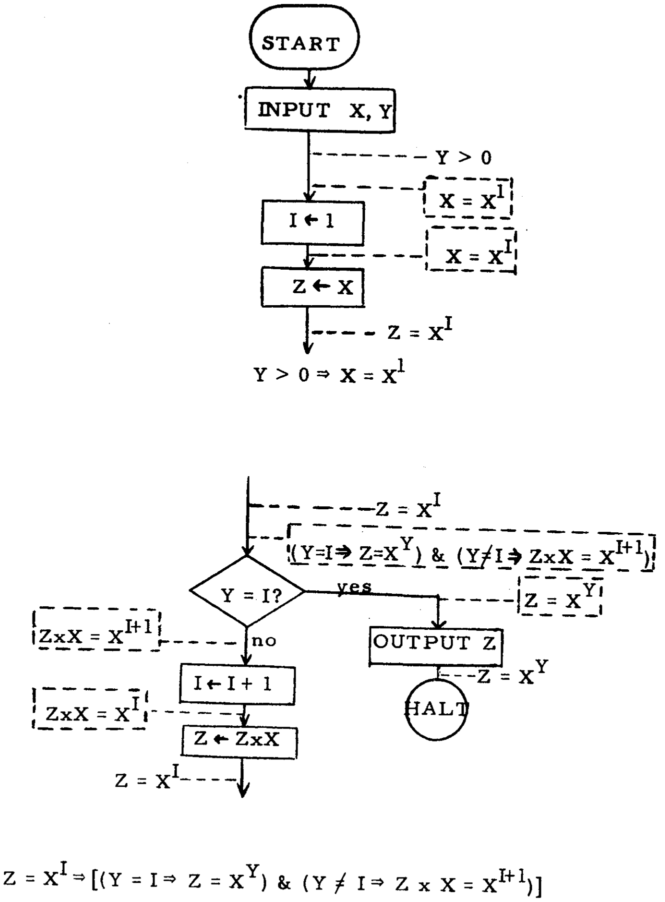


Figure 2—Sectioned flowchart

The careful reader will note that the input assumption $Y > 0$ is not really needed for the proof of either of these theorems. This is because that assumption is really only needed to prove that the program terminates.

Inherent difficulties

This process for proving the correctness of programs is subject to many variations both to handle programming constructs which do not occur in this example and to try to make the proof of correctness more efficient. Full treatments with many examples are available in a recent survey by Elspas, et al.,[3] and in Manna's forthcoming textbook.[4] Some further general comments about the nature of the problem will be made here. Analogous comments could be made about most of the other approaches to proving correctness.

Programs can only be said to be correct with respect to formal specifications or input-output assertions.

There is no formal way to guarantee that these specifications adequately express what someone really wants the program to do.

Given a program with specifications on the input and output, there is probably no automatic way to generate all the additional assertions which must be added to make the proof work. For a human to add these assertions requires a thorough understanding of the program. The programmer should be able to supply these assertions if he is able to formalize his intuitive understanding of the program.

Given a program with assertions in each loop and given an adequate definition of the semantics of the programming language, it is fairly routine to generate the theorems or verification conditions. Several existing computer programs that do this are described below.

The real problem in proving correctness lies in the fact that even for simple programs, the theorems that are generated become quite long. This length makes proving the theorems very difficult for a human or for current automatic theorem provers.

Formalizing the programmer's intuition of correctness

It may not be apparent, but the process of proving correctness is just a formalization into rigorous logical terms of the informal and sometimes sloppy reasoning that a programmer uses in constructing and checking his program. The programmer has some idea of what he expects to be true at each stage of his program (the assertions), he knows how the programming language semantics will transform a stage (generating the assertions in dashed-line boxes of Figure 2), and he convinces himself that the transformations will give the desired result (the proof of the theorem). In this sense proving program correctness is just a way to put into formal language everything one should understand in reading and informally checking a program for correctness. In fact, there is no clear division between the idea of reading code to check it for correctness and the idea of proving it correct by more rigorous means; the difference is one of degree of formality.

One question that should be addressed in this context regards the fact that both the correctness and the halting problems for arbitrary programs are known to be undecidable in the mathematical sense. However, this question of mathematical undecidability should not arise for any program for which there are valid intuitive reasons for the program to be correct.

Confidence in correctness

I hope I have made the point that logical proof of correctness techniques are radically different from testing techniques which are based on executing the program on selected input data in a specific environment. However, I do not want to imply that in a practical situation a proof or anything else can lead to absolute certitude of correctness. In fact a proof by itself does not necessarily lead to a higher level of confidence than might be achieved by extensive testing of a program. From a practical viewpoint there are a number of things that could still be wrong after a proof if one is not careful: what is proven may not be what one thought was proven, the proof may be incorrect, or assumptions about either the execution environment or the problem domain may not be valid. However, a proof does give a quite different and independent view of program correctness, and if it is done well, it should be able to provide a very high level of confidence in correctness. In particular, to the extent that a proof is valid, there should no longer be any doubt about what might happen after allowable but unexpected input values.

MANUAL PROOFS

The basic ideas in the last section have been known for some time. This section describes the practical progress which has been made with manual proofs in the last few years.

The size of programs which can be proven by hand depends on the level of formality that is used. In 1967 McCarthy and Painter[5] manually proved the correctness of a compiler for very elementary arithmetic expressions. It was a formal proof based on formal definitions of the syntax and semantics of the simple languages involved.

Rigorous but informal proofs

A more informal approach to proofs is now popular. This approach is rigorous, but uses a level of formality like that in a typical mathematics text. Arguments are based on an intuitive definition of the semantics of the programming language without a complete axiomatization. Using these techniques a variety of realistic, efficient programs to do sorting, merging, and searching have been proven correct. The proof of a twenty line sort program might require about three pages. It would now be a reasonable exercise for advanced graduate students.

Proofs of significantly more complex programs have also been published. London[6,7] has done proofs of a pair of LISP compilers. The larger compiler is about 160 lines of highly recursive code. It complies almost the full LISP language—enough so it can compile itself. It is a generally unused compiler. It was written for teaching purposes, but it is not just a toy program. Another complex program has been proven correct by Jones.[8] The program is a PL-1 coding of a slightly simplified version of Earley's recognizer algorithm. It is about 200 lines of code. Probably the largest program that has been proven correct is in the work on computer interval arithmetic by Good and London.[9] There they proved the correctness of over 400 lines of Algol code. The largest individual procedure was in the 150-200 line category. A listing of many other significant programs which have been proven correct can be found in London's recent paper.[10]

If a complex 200 line program can now be proven correct by one man in a couple of months, one can begin to think about breaking larger programs into modules and getting a proof of correctness within a few man years of effort. Clearly there are programs for which a guarantee of the correctness of the running program would be worth not man years but many man decades of effort. We had better take a closer look at the feasibility of such an undertaking and what the proof of correctness would really accomplish.

Environment problems

In most existing proofs of program correctness, what has been proven correct is either the algorithm or a high level language representation of the algorithm. With today's computers what happens when the program actually runs on a physical computer would still be anybody's guess. It would be a significant additional chore to verify that the environment for the running program satisfies all the assumptions that were made about it in the proof. Problems with round off errors, overflow, and so forth can be handled in proofs. Good and London,[9] Hoare,[11] and others have described techniques for proving properties of programs in the context of computer arithmetic, but this can make the proof much more complex. Furthermore, to assure correctness of the running program one would have to be sure that all assumptions about the semantics of the programming language were actually valid in the implementation. The compiler and other system software would have to be certified. Finally, this could all be for naught considering the possibility of hardware failure as it exists in today's machines.

Thus, proving the correctness of a source language program is only one aspect of the whole problem of guaranteeing the correctness of a running program. Nevertheless, eliminating all errors from the source language program would certainly go a long way toward improving the probability that the program will run according to specifications.

Errors in the proof

An informal proof of correctness typically is much longer than the program text itself—often five to ten times as long. Thus the proof itself is subject to error just like any other extremely detailed and complex task done by humans. There is the possibility that an informal proof is just as wrong as the program. However, a proof does not have any loops and the meaning of a statement is fixed and not dependent on the current internal state of the computer. To read and check a proof is a straightforward and potentially automatable operation. The same can hardly be said for programs. Despite its potential fallibility, an informal proof would dramatically improve the probability that a program is correct. There is evidence from London's work[7] that a proof of correctness will find program bugs that have been overlooked in the code.

Less rigorous proofs

A person proving a program correct by manual techniques must first achieve a very thorough understanding of all details of the program. This clearly limits manual proof techniques to programs simple enough to be totally comprehended by the program provers. It also means that clarity and simplicity is very important in the program design if the program is to be proven correct. There is another school of thought which places primary emphasis on techniques for obtaining clarity and structure in the program design. Dijkstra[12,13] as long been the primary advocate of this approach. By appropriately structuring the program and by using what is apparently a much less formal approach to proofs, Dijkstra claims to have proven the correctness of his THE operating system.[14] Mills[15] advocates a similar approach with the program being sufficiently structured so an informal proof can be as short as the program text itself.

It is probably true that more practical results can be obtained with less rigorous approaches to proofs, especially in the near future. It is even debatable whether the more rigorous proofs give more assurance of correctness, but the formality does make it more feasible to automate the proof process. Whether or not one feels that the rigorous hand proofs of correctness will have much practical value, they are providing experience with different proof techniques that should be very valuable in attempting to automate the proof process.

AUTOMATING PROOFS OF CORRECTNESS

In proving program correctness the logical statement that has to be proven usually is very long; however, the proof is seldom mathematically deep and much of it is likely to be quite simple. In the example given previously the theorems to be proven were almost trivial. It would seem that some sort of automatic theorem proving should be able to be applied in proving program correctness. This has been tried. So far the results have not been very exciting from a practical viewpoint.

Computer-generated proofs

Fully automatic theorem provers based on the resolution principle generally can prove correctness for very small programs—not much larger than the exponentiation program above. However, Slagle and Norton[16] report that they have obtained fully automatic proofs of the verification conditions for Hoare's sophisticated little program FIND[17] which finds the *n*th largest element of an array. In 1969 King[18] completed a program verifier that automatically generated the verification conditions and then used a special theorem prover based on a natural deduction principle to automatically prove them. This system successfully proved programs to do a simple exchange sort, to test whether a number is prime, and similar integer manipulation programs. The data types were limited to integer variables and one dimensional arrays. Others have experimented with other data types and proof procedures. At the time of this writing I believe that there is no automatic theorem prover which has proven correctness for a program significantly larger than those mentioned.

Automatic theorem provers still cannot handle the length and complexity of the theorems that result from larger programs. Another problem lies in the fact that some semantics of the programming language and additional facts about the application area of the program have to be supplied to the theorem prover as axioms. Automatic theorem provers have difficulty in selecting the right axioms when they are given a large number of them. Even in the minor successes that have been achieved, a somewhat tailor-made set of axioms or rules of inference have been used.

Computer-aided proofs

There are now several efforts directed toward providing computer assistance for proving correctness. This takes the form of systems to generate verification conditions and to do proof checking, formula simplification and editing, and semiautomatic or interactive theorem proving. Unfortunately at this time almost any automation of the proof process forces one into more detailed formalisms and reduces the size of the program that can be proven. This is because the logical size of the proof steps that can be taken in a partially automated proof system is still quite small. Presumably this is a temporary phenomenon. It seems reasonable to expect that we will soon see computer-aided verification systems which make use of some automatic theorem proving and can be used to prove correctness of programs somewhat larger than those that have been proven by hand.

Igarashi, London, and Luckham[19] are developing a system for proving programs written in PASCAL. The verification condition generator handles almost all the constructs of that language except for many of the data structures. Their approach is based on the work of Hoare.[11,20]

Elspas, Green, Levitt, and Waldinger[21] are developing a proof of correctness system based on the problem-solving language QA4.[22] It will use the goal-oriented, heuristic approach to theorem proving which is characteristic of that language.

Good and Ragland[23] have designed a simple language NUCLEUS with the idea that a verification system and a compiler for the language could be proven correct. Both the verification system and the compiler would be written in NUCLEUS and the proofs of correctness would be based on a formal definition of the language. The intent is that the language would then be able to be used to obtain other certified system software.

These three systems give a general idea of the current work going on. A proof-checking system will be described in the next section. Several other interesting systems have been implemented and basic information about them is readily available in London's recent paper.[10]

Long-range outlook

Proofs of correctness are currently far behind testing techniques in terms of the size and complexity of the programs that can be handled adequately. It is very much an open question whether automated proof techniques will ever be feasible as a commonly used alternative to testing a program. Many arguments pro and con are too subjective for adequate consideration here; however, a few comments are in order before one uses the rate of progress in the past as a basis for extrapolating into the future.

Proofs are based on sophisticated symbolic manipulations, and we are still at an early stage of gathering information about ways to automate them. Existing proof systems have been aimed mostly at testing the feasibility of techniques. Few if any have involved more than a couple man years of effort—many have been conceived on a scale appropriate for a Ph.D. dissertation. If and when a cost-effective system for proving correctness becomes feasible, it will certainly require a much larger implementation effort.

Proofs may be practical only in cases where a very high level of confidence is desired in specified aspects of program behavior. With computer-aided proofs one could hope to eliminate most of the sources of error that might remain after a manual proof. As exemplified by the work of Good and Ragland,[23] the verification system itself as well as compilers and other system software should be able to be certified. If the basic hardware/software is implemented with a system such as LOGOS[24] for computer-aided design of computer systems, then there should be a reasonable guarantee that the implemented computer system meets design specifications. With sufficient error-checking and redundancy, it should thus be possible to virtually eliminate the danger of either design or hardware malfunction errors. By the end of this decade these techniques may make it possible to obtain virtual certitude about a program's behavior in a running environment. There are many applications in areas such as real-time control, financial transactions, and computer privacy for which one would like to be able to achieve such a level of confidence.

SOME THEORETICAL FRONTIERS

Proofs of program correctness involve one in a seemingly exorbitant amount of formalism and detail. Some of this is inherent in the nature of the problem and will have to be handled by automation; however, the formalisms themselves often seem awkward. The long formulas and excessive detail may result partially because we have not yet found the best techniques and notation. Active theoretical research is developing many new techniques that could be used in proving correctness. Research in this area, usually called the mathematical theory of computation, has been active since McCarthy's[25,26] early papers on the subject. I feel that practical applications for proofs of correctness will develop slowly unless new techniques for proving correctness can significantly reduce the awkwardness of the formalisms required. This section will describe some of the current ideas being investigated. The topics chosen are those which seemed more directly related to techniques for facilitating proofs of correctness.

Induction techniques for loops and recursion

Proving correctness of programs would be comparatively simple if programs had no loops or recursion. However, some form of iteration or recursion is central to programming, and techniques for dealing with it effectively in proofs have been a subject of intensive study. All the techniques use some form of induction either explicitly or implicitly. The method of inductive assertions described previously handles loops in flowcharts by the addition of enough extra assertions to break every loop and then appeals to induction on the number of commands executed. For theoretical purposes it is often easier and more general to work with recursively defined functions rather than flowcharts. Almost ten years ago McCarthy proposed what he called Recursion Induction[26] for this situation. Manna et al. have extended the inductive assertion method to cover recursive,[27] parallel,[28] and non-deterministic[29] programs. Several other induction principles have been proposed by Burstall,[30] Park,[31] Morris,[32] and Scott.[33] A development and comparison of the various induction principles has been done recently by Manna, Ness, and Vuilleman.[34]

Formalizing the semantics of programming languages

The process of constructing the verification conditions or logical formulation of correctness is dependent on the meaning or semantics of the programming language. One can also take the opposite approach—proving correctness is a formal way of knowing whether a higher level meaning is true of the program. Thus the meaning or semantics of any program in a language is implicitly defined by a formal standard for deciding whether the program satisfies a specification. There is a very close interrelation between techniques for formalizing the semantics of a programming language and proofs of program correctness. Floyd's early work on assigning meanings to programs[1] has been developed especially by Manna[2] and Ashcroft.[35] Burstall[36] gives an alternative way to formulate program semantics in first-order logic. Ashcroft[37] has recently summarized this work and described its relevance.

Hoare,[11,20] Igarashi,[38] de Bakker,[39] and others have worked to develop axiomatic characterizations of the semantics of particular programming languages and constructs. The Vienna Definition Language[40] uses an abstract machine approach to defining semantics, and Allen[41] describes a way of obtaining an axiomatic definition from an abstract machine definition. The axiomatic definition is generally more useful in proofs. Scott and Strachey have developed another approach to defining semantics[42] which is described below.

Work on defining the semantics of programming languages is very active with many different approaches being tried. Those described above are only the ones more closely related to proofs. If any of these ideas can greatly simplify the expression or manipulation of properties of programs, they should have a similar simplifying impact on proofs of correctness.

Formal notation for specifications

Formal correctness only has meaning with respect to an independent, formal specification of what the program is supposed to do. For some programs such specifications can be given fairly easily. For example, consider a routine SORT which takes a vector X of arbitrary length n as an argument and produces a vector Y as its result. With appropriate conventions, the desired ordering on Y is specified by:

$$(\forall i, j)[1 \leq i < j \leq n \Rightarrow Y(i) \leq Y(j)]$$

One also needs a specification about the relation between X and Y. With the property PERM(x) meaning "x is a permutation" and using $\circ$ for functional composition, the following will do:

$$(\exists \mathrm{P})[\mathrm{PERM}(\mathrm{P}) \& Y = X \circ P]$$

Note that the specification allows for any one of many possible algorithms to be chosen—presumably on the basis of efficiency. Yet from an external point of view the specification is complete. If SORT is to be used as part of a larger program, the specifications contain all one may want to know about it.

We can usually define correctness in this way for numeric, mathematical, and other simple programs typically found in program libraries. In fact the causality is largely the other way around: it is worth putting a program in a library to the extent that there is a good way of precisely defining the effects of the program without getting into all the details of its algorithmic implementation.

It would be useful to have a good way of writing formal specifications for a much wider range of computational processes. Parnas has been working on such techniques for formally specifying software modules.[43] His approach does handle error messages, and all side effects have to be carefully formalized.

From a proof of correctness point of view the formalism must have convenient deductive techniques as well as expressive power. First-order predicate calculus has the best deductive techniques, but without extensive definitions and axioms, its expressive power is very poor. For the SORT program above we assumed a definition of permutation, and still the specifications are more obscure than one might desire. For many programs the attempt to define their external effects with the formalism of a fairly standard predicate calculus can lead to extremely long and complex expressions. In particular, proof techniques associated with iteration and recursion have often been awkward when expressed in formal logic. One reason is that recursion and iteration lead to partial functions, that is, functions that may not be defined at all points. There has been a need for the logic that handles undefined values and can be easily used to prove properties of partial functions. Despite many efforts there has been no really successful, agreed-upon logical calculus that dealt with undefined values in a clean and natural way. Some recent work by Scott offers a possible solution to this and other problems.

The work of Scott, Strachey, and Milner

In 1964 Strachey[44] outlined an approach to defining the semantics of a programming language by mapping programs into a mathematical structure built up from a rather small number of precisely specified basic concepts. The approach eliminated any need for an abstract evaluating mechanism. Unfortunately the idea required some mathematical objects (such as self-referential functions) for which there was no firm mathematical foundation.

In 1969 Scott started to work on the underlying mathematical problems. The main breakthrough led to the first matmheatical model of the λ-calculus.[45,46] The work involved the breaking of new ground in both lattice theory and topology. Function spaces are considered as lattices by using the "is consistent with and less defined than" relation on partial functions for the lattice partial ordering. It is then possible to define a logic with a fairly natural induction scheme which seems to have great generality and ease of expression for proving properties of recursively defined functions.

Scott's techniques allow for the construction of a universe of computable mathematical functions which is sufficiently general so that it should be possible to define the meaning of any program by associating with it a specific function in this universe.[42,47] The semantics of a program are thus defined mathematically in terms of a limited number of basic mathematical concepts and not in terms of the result of a calculation on a machine. The semantical function that makes the association is defined recursively on the syntax of the program. The mathematical universe is sufficiently general so that the semantical function itself exists within the universe.[33]

The practicality of this approach has yet to be determined, but it seems to hold out the hope of a much less cumbersome way to formalize semantics. This mathematical approach to semantics may enable one to abstract from the arbitrary choices a great amount of extraneous detail that is typical of program implementations. The trick, of course, is to abstract from the right detail without losing important properties of the program.

Milner[48] has implemented a mechanical proof checker for a logic of computable functions based on some of the work of Scott. The implementation includes extensive simplification mechanisms and an interactive goal setting structure. Milner and Weyhrauch have used the logic to formalize semantics,[49,50] to prove simple program correctness,[50] and to give a mechanical proof of compiler correctness based on formally defined semantics.[51] The proof checker is still limited to proving properties of rather small programs; however, expressing formal properties of programs does seem to be simplified. The expression simplification mechanisms have also been useful.

The nature of this and other active theoretical research indicates that there may soon be techniques which will significantly simplify the problem of proving program correctness.

INTEGRATING PROOFS WITH PROGRAM DESIGN

Proving program correctness has usually been done after a program is written. An alternate approach is to integrate the proof with the program design. This approach provides some hope that proofs might eventually help to organize and simplify the program production process. A proof of correctness will greatly increase the amount of formalism that must be dealt with. However, if a proof can be integrated into the design and writing stages, it should eliminate most of the need for debugging and may alleviate the problems of documentation and maintenance. Floyd[52] has envisioned an automated verification system such that a programmer can interact with it in real time as he is writing his program.

Hoare's proof of correctness for his program FIND[53] was done in a top down way with the program and the proof evolving simultaneously. Jones in his proof of Earley's recognizer algorithm[8] exemplified a process he calls the formal development of correct algorithms. It is the longest published example of how a proof might discipline program design.

Throughout the development of the algorithm Jones uses a special formal notation related to the Vienna Definition Language, he does not introduce an ordinary programming language until the very end. With this notation he was able to give a formal, non-procedural specification for a recognizer in about half a page. He then develops the algorithm by stages while at each stage extending a proof that the partially developed algorithm will meet the specifications. At each stage the proof depends on formally expressed assumptions about the undeveloped part of the algorithm.

At the present time the amount of formalism required for the proof tends to overwhelm the program design effort. Nevertheless, this approach appears to make proofs of correctness somewhat more practical in an actual programming environment.

Automatic program synthesis

Rather than writing both specifications and a program, one might want to let the computer create the program and thus be responsible for its correctness. One technique for automatic program synthesis is closely related to techniques for proving correctness. One proves that there is an output satisfying the specifications and then extracts a program from this proof.[54,55] By using induction in the proof, it is possible to construct programs with loops. Manna and Waldinger have given several examples of this.[56]

While automatic program synthesis would be more useful than proving correctness, automatic synthesis requires a much more difficult proof. Since techniques for generating the required proofs are the major unsolved problem in this whole area, this form of automatic program synthesis is a more long-range goal than proofs of correctness.

CONCLUSION

Work on proving properties of programs has progressed to the point where one can argue whether there will soon be useful results. It is mostly a matter of what one means by "useful".

The software engineer who is worried about large programming projects will find current proof techniques hopelessly inadequate for all the large scale problems that are the center of his concern. Even for small modules he will probably find that test methods are more cost-effective than rigorous proofs. One should be able to obtain very great confidence in the correctness of a moderate-sized program if the level of talent and resources that would be necessary for a rigorous proof were devoted to reading and testing the program. Considering the time it normally takes for research results to work their way into practical applications, I would expect that it will be at least three or four years before this situation changes significantly.

Within the next three or four years, less rigorous techniques for structuring, understanding, and checking a program may become widely used. More rigorous proof techniques could be useful on small critical modules where adequate confidence cannot be achieved by other means. In this case it may be worth the additional cost of a proof to obtain an independent evaluation of correctness.

While most work on proving correctness has been for programs written in higher level languages, the most useful early applications may occur either for algorithms at the hardware or microcode level or for the calling structure at the highest level in the design of a large program. In both cases there is a high priority on correctness, and one would like to be assured of correctness long before testing becomes possible.

If current research on simplifying and automating the proof process can significantly reduce the difficulty of proving correctness, then in a few years proofs may be commonly used on small critical modules. Gradually the proof techniques could then be extended to larger programs so that they can be more useful in implementing very reliable systems. It is unlikely that proof techniques will be cost-effective for routine programs within this decade, but the potential is there for eventually revolutionizing the software marketplace.

REFERENCES

1 R W FLOYD
Assigning meanings to programs
Proceedings of a Symposium in Applied Mathematics Vol 19 Mathematical Aspects of Computer Science American Mathematical Society 1967 pp 19-32

2 Z MANNA
The correctness of programs
Journal of Computer and System Sciences Vol 3 No 2 May 1969 pp 119-127

3 B ELSPAS K N LEVITT R J WALDINGER A WAKSMAN
An assessment of techniques for proving program correctness
Computing Surveys Vol 4 No 2 June 1972

4 Z MANNA
Introduction to the mathematical theory of computation
McGraw Hill Book Co Inc to be published

5 J McCARTHY J PAINTER
Correctness of a compiler for arithmetic expressions
Proceedings of a Symposium in Applied Mathematics Vol 19 Mathematical Aspects of Computer Science American Mathematical Society 1967 pp 33-41

6 R L LONDON
Correctness of a compiler for a LISP subset
Proceedings of an ACM Conference on Proving Assertions about Programs
SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 Jan 1972 pp 121-127

7 R LONDON
Correctness of two compilers for a LISP subset
Artificial Intelligence Memo 151 Stanford Univ Oct 1971

8 C B JONES
Formal development of correct algorithms: An example based on Earley's recognizer
Proceedings of an ACM Conference on Proving Assertions about Programs
SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 Jan 1972 pp 150-169

9 D I GOOD R L LONDON
Computer interval arithmetic: Definition and proof of correct implementation
Journal of the ACM Vol 17 No 4 Oct 1970 pp 603-612

10 R L LONDON
The current state of proving programs correct
Proceedings of the ACM Annual Conf ACM 1972

11 C A R HOARE
An axiomatic basis of computer programming
Communications of the ACM Vol 12 No 10 Oct 1969 pp 576-583

12 E W DIJKSTRA
Notes on structured programming
Technische Hogeschool Eindhoven August 1969

13 E W DIJKSTRA
A constructive approach to the problem of program correctness
BIT Vol 8 1968 pp 174-186

14 E W DIJKSTRA
The structure of the "THE" multiprogramming system
Communications of the ACM Vol 11 No 5 May 1968 pp 341-346

15 H D MILLS
The complexity of programs
Proc of SIGPLAN Symposium on Computer Program Test Methods Prentice-Hall to appear

16 J R SLAGLE L M NORTON
Experiments with an automatic prover having partial ordering rules
Heuristics Laboratory, National Institutes of Health 1971

17 C A R HOARE
Algorithm 65, find
Communications of the ACM Vol 4 No 7 July 1961 p 321

18 J C KING
A program verifier
PhD Thesis Carnegie-Mellon University Sept 1969

19 S IGARASHI R L LONDON D LUCKHAM
Private communication

20 C A R HOARE
Procedures and parameters: An axiomatic approach
Symposium on Semantics of Algorithmic Languages
E Engeler ed Springer-Verlag 1971 pp 102-116

21 B ELSPAS M W GREEN K N LEVITT
R J WALDINGER
Research in interactive program-proving techniques
Stanford Research Institute Report May 1972

22 J F RULIFSON R J WALDINGER
J A DERKSEN
A language for writing problem-solving programs
Foundations of Information Processing IFIP Congress 71
North Holland Publ Co 1971 pp 111-115

23 D I GOOD L C RAGLAND
NUCLEUS—A language for provable programs
Proc of SIGPLAN Symposium on Computer Program Test
Methods Prentice-Hall to appear

24 C W ROSE
LOGOS and the software engineer
These Proceedings

25 J McCARTHY
Towards a mathematical science of computation
Proc of IFIP 1962 C M Popplewell ed North Holland Publ
Co pp 21-28

26 J McCARTHY
A basis for a mathematical theory of computation
Computer Programming and Formal Systems P Braffort
D Hirschberg eds North Holland Publ Co Amsterdam 1963
pp 33-70

27 Z MANNA A PNUELI
Formalization of properties of functional programs
Journal of the ACM Vol 17 No 3 July 1970 pp 555-569

28 E A ASHCROFT Z MANNA
Formalization of properties of parallel programs
Machine Intelligence 6 Edinburgh Univ Press 1971

29 Z MANNA
The correctness of non-deterministic programs
Artificial Intelligence Vol 1 No 1 1970

30 R M BURSTALL
Proving propertis of programs by structural induction
Computer Journal Vol 12 1969 pp 41-48

31 D PARK
Fixpoint induction and proofs of program properties
Machine Intelligence 5 B Meltzer D Michie eds Edinburgh
Univ Press 1969 pp 59-78

32 J H MORRIS
Another recursion induction principle
Communications of the ACM Vol 14 No 5 May 1971
pp 351-354

33 D SCOTT
The lattice of flow diagrams
Symposium on Semantics of Algorithmic Languages
E Engeler ed Springer-Verlag 1971 pp 311-366

34 Z MANNA S NESS J VUILLEMIN
Inductive methods for proving properties of programs
Proc of an ACM Conference on Proving Assertions about
Programs SIGPLAN Notices Vol 7 No 1 and SIGACT News
No 14 Jan 1972 pp 27-50

35 E A ASHCROFT
Mathematical logic applied to the semantics of computer programs
PhD Thesis Imperial College London 1970

36 R M BURSTALL
Formal description of program structure and semantics in first order logic
Machine Intelligence 5 Edinburgh Univ Press 1970 pp 79-98

37 E A ASHCROFT
Program correctness methods and language definition
Proc of an ACM Conference on Proving Assertions about
Programs SIGPLAN Notices Vol 7 No 1 and SIGACT News
No 14 Jan 1972 pp 51-57

38 S IGARASHI
Semantics of ALGOL-like statements
Symposium on Semantics of Algorithmic Languages
E Engeler ed Springer-Verlag 1971 pp 117-177

39 J W DeBAKKER
Axiom systems for simple assignment statements
Symposium on Semantics of Algorithmic Languages
E Engeler ed Springer-Verlag 1971 pp 1-22

40 P WEGNER
The Vienna definition language
Computing Surveys Vol 4 No 1 March 1972 pp 5-63

41 C D ALLEN
Derivation of axiomatic definitions of programming languages from algorithmic definitions
Proc of an ACM Conference on Proving Assertions about
Programs SIGPLAN Notices Vol 7 No 1 and SIGACT News
No 14 Jan 1972 pp 15-26

42 D SCOTT C STRACHEY
Toward a mathematical semantics for computer languages
Proc of the Symposium on Computers and Automata
Microwave Research Institute Symposia Series Vol 21
Polytechnic Institute of Brooklyn 1971

43 D L PARNAS
A technique for software module specification with examples
Communications of the ACM Vol 15 No 5 May 1972
pp 330-336

44 C STRACHEY
Towards a formal semantics
Formal Language Description Languages for Computer
Programming T B Steel Jr ed North Holland Publ Co
Amsterdam 1966 pp 198-220

45 D SCOTT
Outline of a mathematical theory of computation
Proc of the Fourth Annual Princeton Conference on
Information Sciences and Systems 1970

46 D SCOTT
Lattice theory, data types and semantics
Formal Semantics of Programming Languages R Rustin ed
Prentice-Hall 1972 pp 65-106

47 D SCOTT
Mathematical concepts in programming language semantics
AFIPS Conference Proc Vol 40 SJCC 1972 pp 225-234

48 R MILNER
Logic for computable functions; Description of a machine implementation
Artificial Intelligence Memo 169 Stanford Univ May 1972

49 R MILNER
Implementation and applications of Scott's logic for computable functions
Proc of an ACM Conference on Proving Assertions about
Programs SIGPLAN Notices Vol 7 and SIGACT News
No 14 Jan 1972 pp 1-6

50 R W WEYHRAUCH R MILNER
Program semantics and correctness in a mechanized logic
Proc USA-Japan Computer Conference Tokyo 1972

51 R MILNER R W WEYHRAUCH

Proving compiler correctness in a mechanized logic
Machine Intelligence 7 Edinburgh Univ Press 1972

52 R W FLOYD
Toward interactive design of correct programs
Invited Papers IFIP Congress 71 North Holland Publ Co 1971 pp 1-4

53 C A R HOARE
Proof of a program: FIND
Communications of the ACM Vol 14 No 1 Jan 1971 pp 39-45

54 C C GREEN
The application of theorem proving to question-answering systems
PhD Thesis Stanford Univ 1969

55 R J WALDINGER R C T LEE
PROW: A step toward automatic program writing
Proc International Joint Conf on Artificial Intelligence Washington DC 1969

56 Z MANNA R J WALDINGER
Towards automatic program synthesis
Communications of the ACM Vol 14 No 3 March 1971 pp 151-165

Supercomputers for ordinary users*

by DAVID J. KUCK

University of Illinois at Urbana-Champaign
Urbana, Illinois

INTRODUCTION

The best way to begin this paper is by explaining its title. I take "supercomputers" to mean those computers which are the biggest, fastest and most complicated available. And by "ordinary" users I mean "nonsuper" users, i.e., people who have had at most an introductory programming course. The point of this paper is to discuss how computer system design and organization could (and I believe should) proceed in the next ten or twenty years.

Briefly put, I think that by careful analysis of users' algorithms, designers could produce computers which are much more cost effective than present machines. It is also the designers' responsibility to consider new uses for machines, particularly in the non-numerical file processing area, and thus make the benefits of super-computers more directly available to ordinary users.

This paper does not present some machine organization as "the one best way." In spite of the many papers, debates and advertisements which have been carried out in this spirit, it should be obvious that such an attitude is absurd. Clearly, there are many different computer/software configurations extant which serve quite different sets of users in reasonably acceptable ways. In fact, it is not difficult to find people who believe that the architecture of computers is a subject which has reached the end of its line—at least for many applications. "General purpose computers have been built and improved for some 25 years," they say, "and the real problem now is getting better software." They have a point. But, unfortunately, they are viewing and discussing an effect rather than its causes.

It would be impossible to discuss these causes in any detail in this short paper, even if I knew what all of them were. But it is safe to lump many of them as follows: Too many layers of software are being used in an attempt to make general purpose computers appear as a variety of special purpose machines. Does it really make sense to force basically the same machine organization to serve for large numerical calculations, interactive airline reservation systems, and system simulation studies? As further reinforcement, consider the fact that central processors typically account for 20 percent or less of the total hardware bill paid by most computer center managers today.[21] Primary and secondary memories account for over 50 percent and other peripheral devices make up the remainder. And since hardware accounts for at most about half of the overall costs (software, people, and facilities are the rest), perhaps less than 10 percent of the total cost is in the central processor. Is the real point of central processor unit design to encourage more research on hardware and software for memory hierarchies? Rather, it would seem more reasonable to design various processors which are much faster because they contain many more gates than present machines, which are tailored to different kinds of users, and which tend to reduce software needs.

I believe that the architecture and organization of computer systems is a subject which is just now beginning to be studied properly. As a result of current studies, computer systems could begin to be very different from their present form, say, 20 years hence. Sensitive people now have enough intuition about the "rights" and "wrongs" of enough sufficiently different hardware and software ideas to make this possible. We should be able to write programs which serve as computer system design aids using various formalizations of this intuition. Thus, while it has not been possible to find "the one best way" in terms of one machine to satisfy all users, it should be possible to find comprehensive design procedures which yield good machines for particular classes of computations. This would effectively automate or eliminate the intellectual interest from whole classes of computer system design. Since it may

* This work was supported in part by NSF Grant GJ 27446.

be assumed that technology and user algorithms will keep changing forever, such design procedures should be relatively independent of those matters.

Now let us return to the "ordinary" users of our title. These ultimate consumers of computation and its products have almost no idea about what kind of computer they *should* be using. Indeed, in view of the present state of affairs sketched above, probably no machine designer knows either. But if steps were taken to automate the design of entire computer systems, then it should be possible to incorporate these users' needs by analysis of the algorithms they wish to run. My point is that the best possible "consumer advocate" for ordinary (and all other) users is the computer system designer.

In order to make our discussion concrete, the following list is presented and will be elaborated in the remainder of this paper:

Computer designer's goals and responsibilities

- Computerism
  1. Reduce the use of high cost components and subsystems in a machine.
  2. Increase the use of low cost components and subsystems in a machine.
  3. Maximize and balance data flow rates throughout a system by introducing concurrency.
- Consumerism
  4. Measure user algorithms to determine system design.
  5. Make machines extensible in their capabilities.
  6. Incorporate new processing features into hardware to replace programs.

This distinction between "computerism" and "consumerism" is somewhat arbitrary, but we wish to emphasize the designers' two responsibilities. "Cost" in the above refers to some criterion function incorporating both time and money.

SYSTEM COST EFFECTIVENESS

There is no single accepted definit'on of "cost effectiveness", but, like the weather, everyone talks about it and this paper is no exception. First, let us consider effectiveness in terms of speed. Users are interested in fast job completion in batch systems and fast response in interactive systems. Multiprogramming and multiprocessing are two important architectural techniques which have been introduced as aids in fulfilling both of these goals for ordinary users. We shall discuss these below, ignoring the pending large parallel and pipeline machines since they are generally not intended for ordinary users. In the following, "primary memory" means all random access memories including bulk core storage.

Since multiprogramming causes several users and some system software to be in primary memory at the same time, it is usually possible to keep several parts of the machine (e.g., I/O and CPU) busy at the same time. This introduction of a number of concurrent tasks speeds up the overall completion of an ensemble of tasks. Since several users are in primary memory, it is also possible to interrupt each of them after a short time and output the results of some progress on their job—hence speeding up system response time. But the penalties paid are well-known. The two most important are very large random access memories and very large system programs which may lead to more CPU cycles being used by the system than by the users. Demand paging is an attempt to decrease primary memory size, but it leads to even more system overhead. Several system redesign approaches have been attempted. An intuitively simple and good one is to dump from primary memory all of a user's pages when he has a page fault and then obtain the missing page and merge it with the dumped set before restarting him. This was attempted in the BCC 500 computer, which has not become operational.

Multiprocessing has also been introduced in an attempt to speed up machines. Here several programs share a set of parallel memories and processors. Again, there is a system program overhead to decide what to do next. Also there are problems of memory conflicts which prevent all but one program from proceeding when several must access one memory unit at the same time. This can degrade the system to the effectiveness of a single processor. Finally, there are problems of primary memory size. Unless it is increased·over that of a monoprocessor a large amount of swapping or page faulting will degrade the system. Attempts to use more than three or four processors in this way have generally been unsuccessful. Two processors are usually as effective as about 1.5 or 1.7 processors and in one fortunate case four processors were as effective as three.[17] But in all multiprogramming and multiprocessing situations, it is "the mix" of jobs which appear simultaneously that determines the success or failure of the ystem's performance. Obviously, no designer can do nything about such random phenomena except in a statistical way. For a survey of some details of the above matters, the reader is referred to References 7 and 12.

Later we will give some details of a new scheme for delivering the performance demanded by batch and interactive users. In this scheme batch users are given fast service by executing their programs using highly concurrent processing. Many processors and memory units are involved. Interactive users are satisfied by partitioning (space sharing) the overall machine and simultaneously executing a number of them. Each such active user's program may still use a number of processors and memories, however.

What is new about this proposal? Superficially nothing; all of the general ideas are old ones. Several basic points are new, however. They include techniques for the analysis of ordinary programs to run them on such a machine. We will further discuss these below. But we must now deal with the "cost" part of cost effectiveness.

In the Introduction under Computerism, points 1 and 2 were about system cost. These are both obvious and well-known. In the last five years many meetings have dealt with the future of computers and every one worth its salt has had a paper entitled "The Impact of LSI on . . .," where the reader may insert his favorite computer part for the ellipsis. And indeed large-scale integration, when it arrives, should decrease cost and size while increasing reliability and speed. But too often people rejoice about such a thing as being able to afford 5 times as much random access memory and hence treat the symptoms rather than the causes of problems with present computers. Our point is that the availability of large amounts of logic should be regarded as a chance to put more processing back into computers.

It should be noted that the equivalent of about ten or twenty thousand gates is about the most that can reasonably be used in a fast, standard arithmetic and logic unit. Furthermore, it should be observed that the speed of arithmetic in such units is nonlinearly expensive. In other words, anticipating carries in d bit arithmetic units requires about $\alpha d \log_2 d$ gates ($\alpha \leq 10$) to add two numbers in $0(\log_2 d)$ time while less than 10 gates can do the job in time $0(d)$. Of course, fixed point addition is not equivalent to a modern arithmetic unit, but the trend is clear.

We draw two conclusions from the above. The first is that functions which have not been incorporated in traditional arithmetic and logic units (and control units as well) should now be considered for inclusion. Thus, more gates would lead to higher speeds in various new applications and we mention several later.

Our second conclusion is that in the traditional area of arithmetic, when many concurrent (parallel or pipeline) operations may be performed one must rethink the individual arithmetic algorithms. For example, it is not difficult to design a medium speed arithmetic unit with about 1000 gates and call this an LSI component. If sufficiently many components of this type could be operated simultaneously, high system speed could be achieved at low cost. In general, one must consider various kinds of logic, various numbers of bits per step, and various numbers of words to process concurrently using parallel and pipeline techniques. As an exercise, the reader is invited to consider the range of design possibilities for an inner product processor (or component) with high speed and low cost. If a range of logic speeds and costs, the number of simultaneous arithmetic operations, and the number of bits per step are all design parameters many new possibilities arise. The high speed of such designs results from concurrent processing rather than relying on state of the art technology as has often been the case. Low cost in dollars, development time and headaches would result from the use of conservative, well understood circuits. Thus, good cost effectiveness could be obtained through proper system organization.

ANALYSIS OF PROGRAMS

That hardware and software designers should study user programs is not a new idea. Most modern machines are built on the basis of some such analysis and simulation. However, here we are arguing for new kinds of analyses which lead to relatively more processor and less memory cost while speeding up the system's performance without using faster parts.

Various classes of programs should be studied to determine for some general machine assumptions at what theoretically maximum speed they may be processed. This is a study of the semantics and pragmatics of programming languages, not the syntax. Then the practicalities of actual machine design should lead to proposed systems. The following outlines our steps in this direction.

First, consider the set of all FORTRAN programs. Ignoring input/output statements, the executable statements in a FORTRAN program can be lumped into three kinds of blocks: DO loops, blocks which are outside DO loops and which contain only assignment statements, and clusters of (one or more) IF statements with only a few intervening assignment statements. We have developed a number of algorithms and implemented an operation level analyzer[10,15,18] which breaks FORTRAN programs into these three kinds of blocks. Using the analyzer, we have measured a number of parameters for a collection of FORTRAN programs totalling several thousand statements. Some of these programs were dominated by DO loops and others had no DO statements at all. All were short programs (average less than 50 statements). On this basis we observe that a machine capable of executing 16 operations simultaneously could be effectively used in processing simple ordinary FORTRAN programs. For some of the programs we analyzed and for many large computations, several hundred to a few thousand operations could be performed simultaneously. We emphasize that no multiprogramming is being assumed here. While traditional multiprogramming and multiprocessing have led to speed improvements of one or two binary orders of magnitude, our simple programs enjoyed speed-ups of from one to five binary orders of magnitude and bigger speedup could be achieved for bigger programs.

The analysis of programs need not be restricted to empirical study. In fact, bounds on the time required by certain classes of programs can be obtained as analytical functions of parameters of programs which may be measured statically. Thus, designers could know how well they were doing in some absolute sense. For example, it has recently been shown by demonstrating an algorithm,[1] that any arithmetic expression containing only the add, subtract and multiply operators can be evaluated in less than 2.5 $\log_2 n$ steps if the original expression contains n variables and constants. This upper bound is quite close to the lower bound of $\log_2 n$ steps (assuming two argument fan-ins). Note that this results in a speedup of about $n/2.5 \log n$.

While the above remarks were restricted to FORTRAN programs, any FORTRAN-like program (e.g., ALGOL or parts of PL/I) will obviously yield to a similar analysis. In fact, programs in languages of a quite different nature from FORTRAN may be handled by these techniques. For example, all of the important blocks (routines) of GPSS, a discrete system simulation language, have been so analyzed.[4] Roughly speaking, any program which may be broken into one or more of the three kinds of blocks mentioned earlier (pure assignment statement, loop, conditional jump tree) could be analyzed in this way. In order to achieve the speedup mentioned above certain machine organization features must be introduced.

MACHINE ORGANIZATION

Now we turn to a discussion of the organization of a class of machines. The members of this class are all made from the same building blocks, but the configurations vary in size and organization to suit users' programs. We break the computation process into three kinds of operations: accessing data, aligning data and processing data. To achieve speed, we assume that all three are carried out simultaneously in a pipeline fashion each driven by its own control unit. Furthermore, we assume that each of the three exploits pipeline and/or parallel organizations of parts to achieve high speed.

At this point we expand point 3 under Computerism in the Introduction. Subject to the constraints of the computations to be performed:

(1) The data path bandwidths should be matched everywhere in the system and maximized.
(2) Either pipeline or parallel operations can be used to match data path bandwidths.
(3) Long pipelines introduce long delays in transient data flow while large parallel arrays introduce data alignment problems.
(4) Sufficient control unit lookahead and local (cache) memory should be provided so as to maintain a steady state flow of data.

In the following we assume that these are followed and that steady state data flow through the computer system may be assumed to be a reasonable measure of its performance.

By *accessing* data we mean the operations of fetching and storing in the primary memory. Assume that n operands are needed in a unit time step by the rest of the machine. A limiting case is where one memory unit with a data rate n times faster than the rest of the system is used. Any pattern of n words may be accessed in unit time simply by using an index register to access various locations. Alternatively, if a parallel set of memory units is used, access conflicts arise if several data elements needed simultaneously are stored in the same memory unit. While this may be a serious problem in current multiprogrammed machines, if only one program is running, then array accessing of the kind actually found in real FORTRAN programs is possible without conflicts. Some theoretical limitations and a proposed class of memories together with useful partitions which may be accessed in one memory cycle are discussed in Reference 2. Such a memory works well for the array fetching requirements of DO loops. But blocks of assignment statements or IF blocks require random patterns of scalars. However, the total number of distinct scalars is small, probably less than 4096 in any FORTRAN program. Thus if inexpensive, slower technology were used elsewhere in a machine, the scalar memory could be a relatively expensive, high speed semiconductor memory which operates as discussed above by using an index register to provide any pattern of data required.

By *alignment* we mean the bringing together of proper pairs of numbers for calculation. In a system containing many memories and processors, the arguments needed by one processor on some step may come from that processor's registers (or cache), some memory unit, some other processor or a control unit (for program constants). The problem of designing alignment networks is very important, dependent on the details of computations to be performed and not easy to solve for large systems. It should be noted that using the kinds of memories discussed above leads to alignment problems which are relatively simple. We have studied various aspects of the problem in References 11, 14 and 13, in which a number of options are discussed.

Processing data refers to those operations which either transform the data into a different form or examine the data to determine future actions. We raise three points here and discuss file processing later. In order to evaluate IF blocks quickly, tree processing logic has been designed which evaluates a many way jump in a few clocks.[5] Its effectiveness for FORTRAN is discussed in Reference 15 and for GPSS in Reference 4. Our second point is that in the evaluation of FORTRAN-like programs, the quick execution of supplied functions is obviously desirable. A uniform scheme for this is described in Reference 6, and has been implemented experimentally using a small microprogrammed processor. This leads to our third point which concerns pipelining. Since the speed improvement gained by pipelining depends on the number of levels in the pipeline, it is in one's best interest to choose more time-consuming operations for pipelining although this must be balanced against design principle (3) mentioned above.

If a machine is cleanly broken into memory, alignment, and processing sections, several advantages accrue. One is that a great variety of configurations is possible. For example, machines which need complicated alignment networks can have them, but still use the same processors as machines with simple interconnection problems. Similarly, programs with many jumps can have a more powerful tree processor of the kind mentioned above.

Another advantage of breaking the design into memory, alignment and processor sections is that with care the size and hence the effective speed of the machine can be increased as the work load increases. For example, assume a 16 processor cluster of processors, memories and alignment networks were a basic building block. Using combinations of these, systems containing say 64, 256, etc., processors could be built to provide high speed for larger and larger jobs. Also, in the large configurations 16 unit clusters could still be used in a space shared way to run many small jobs simultaneously and give quick response. The details of this are somewhat messy but for the kinds of systems being outlined here some details are given in Reference 11. Generally, this is a very important matter. It allows users not to worry (as soon) about reprogramming for a completely different machine organization. Indeed this kind of idea was probably the most important in the original 360/series announcement. Note that point 5 under Consumerism in the Introduction raised this subject.

MEMORY HIERARCHIES

So far we have outlined some ideas about how more processing could be done simultaneously in computers. This exploits cheap parts by using more of them. We have discussed how, by detailed program analysis, effectively high computation speeds may be achieved and at the same time primary memory size may be reduced by monoprogramming instead of multiprogramming. The latter follows from the fact that only one user's program and less system program must be resident in primary memory if swapping or a BCC 500 type paging scheme is used. Now we go a step further in reducing primary memory size by exploiting a memory hierarchy for a monoprogrammed machine. The main motivation for memory hierarchies is to reduce costly fast primary memory in favor of less expensive per bit slow secondary memories. Many technologies are available and more are likely to appear. But techniques for effectively using memory hierarchies are not well understood.

Earlier we mentioned demand paging schemes for multiprogrammed machines which attempt to exploit memory hierarchies. An alternative is anticipatory paging for a monoprogrammed machine. Suppose that some program makes repeated sweeps over a large data base, one much too large for primary memory. Furthermore, suppose that by some kind of analysis the program may be broken into (one or more) parts which have reasonably predictable running times. If at the beginning of one sweep over the data it is known what sequence of I/O blocks the program will require on the next sweep, it is possible to effectively mask all rotational latency of a disk or drum on which this data is stored. This is possible regardless of what reordering of the data is required from sweep to sweep and is accomplished by using a rather small buffer memory. For more details see Reference 9.

The above ideas were motivated by the difficulties of large partial differential equation and matrix computations. In Reference 8 several illustrative examples are given from these areas but their applicability is not restricted to such problems. Indeed, there seems to be a big need for such techniques in the area of business data processing and related areas. This leads us to our next topic.

FILE PROCESSING

The exploitation of memory hierarchies and new kinds of processor organization in wide areas outside of FORTRAN-like programs seems to be a very ripe possibility. In the area of what might be called COBOL-like programs, existing systems which do not work well are easy to find.

These computations are often dominated by I/O operations on current machines.[3] It is likely that the analysis of such programs would indicate that what processing is done could often be speeded up by the introduction of more simultaneous processing. In this same vein, demands for management information systems, information retrieval systems, computer-aided education and so on appear constantly. It is easy to think of new applications which might in fact lead to leisure time edification of ordinary users who know practically nothing about computer use. The ability to browse through newspaper files to get background when a news event breaks, the ability to ask about various aspects of history (e.g., music, politics, science) in quick succession, the ability to consult an expert body of details about a technical hobby (e.g., photography), this is a quick list of things that ordinary users should be able to get from supercomputers in twenty or so years. These ordinary users should include managers, students, people at home, scientists and so on.

Point 6 under Consumerism in the Introduction was aimed at matters like this. Such areas are underdeveloped today but with a concentrated push by designers should not remain so. Of course, the reader should keep in mind that Vannevar Bush pointed all this out some 25 years ago in "As We May Think," but in fact technology has come a long way since then.

Indeed, technology does not seem to be the problem now. It would be easy to do searches by streaming data from many heads of a rotating memory through some kind of associative processor (perhaps combining associative hardware and hash addressing). This processor could be of the type described in Reference 20 or a set of coordinated simple processors (mounted on the read heads or externally).[16] In fact, IBM built HARVEST in this spirit over 10 years ago, but a commercial version was never made available. The details of how to use such a system in various contexts and how the system configuration should be stylized to serve particular problem areas best is the central problem. We have been exploring such problems experimentally with a small microprogrammed processor and a disk.[19,22] While we do not have the problem solved, the following are some observations.

First, we remark that systems which use abstracts only or some kind of (nonexhaustive) indexing are crippled from the outset by not being aware of everything in the source material. On the other hand, full text searching for certain word co-occurrences can be overwhelmingly time consuming in some case; even for feasibly powerful hardware systems. Thus it seems that while the full text must be available for machine searching, concordances with pointers should be associated with the text at various levels. This allows the range of detailed searching to be narrowed beforehand.

After certain portions of the full data base have been selected by the machine as possibly interesting, it seems more reasonable to narrow the set of hits by allowing more detailed user interrogation and search than to rely on complicated, previously designed filtering programs. Thus, the system must be interactive and the user must be allowed to move his context of search from the whole data base, to various previously selected parts and back again.

We conclude by repeating several points. First, there are many different kinds of bulk storage devices available today. Interesting memory hierarchy management schemes are beginning to appear. Overall, hardware systems are easy to configure but useful ones require careful algorithm development. The study of COBOL-like programs may be useful in this respect, and analysis similar to that discussed for FORTRAN-like programs may yield interesting results about new processor configurations. But organizing the systems, their data and algorithms remains an important and hard design problem.

SUMMARY

This paper outlines some of the responsibilities machine designers have to ordinary users. If sufficiently complex analytical methods were used in the measurement of users' programs and the design of computer systems, the resulting machines could be much more cost effective than present ones. Low system cost can be achieved by avoiding the fastest possible hardware and by exploiting memory hierarchies. High effectiveness can be achieved in several ways, including the implementation of high level functions in processors and by performing many operations concurrently. We did not mention the problem of replacing parts of operating systems by dedicated microprogrammed processors in control units, an area that deserves careful study. Another important effectiveness improvement would be to make the benefits of machines available to more ordinary users through new kinds of processing.

Some details were sketched of how logic design might be changed if more degrees of freedom were introduced by more concurrent processing. At this point the design of computer systems moves a step closer to the doors of the semiconductor manufacturers. The results of the analysis of programs could be used to combine chips properly, or going one step further they could be used to constrain the design of the chips themselves.

In short, we have presented arguments and methods for achieving a better match between machines and the computations of users. Finally, we remark that careful planning should go into all systems design to allow the easy measurement of as many useful parameters as possible to study a machine after it is built, and to aid in subsequent designs.

CONCLUSION

It is very easy (and hence quite common) for university professors to go around waving their arms and pontificating about how things should be. Every sensible computer person knows that it is not such people who determine what machines will come into common use, but rather "market forces" which do. These mystical processes primarily involve actual users of computers, their bosses; salesmen of computers, their bosses; managers of computer manufacturing; and boards of directors. Paper and real machines conceived in universities, by users, and even in the back rooms of computer manufacturing establishments, are ignored. "You have to give users something they can understand."

In the last few years there has been a tendency for computer manufacturers in the U. S. to merge with each other and two large corporations got out of the business. If the trend continues we could perhaps expect less rather than more variety and imagination in available machines. Recall that before 1920 there were perhaps several hundred different automobile varieties available in the U. S., powered by steam, electricity, gasoline, etc. By the late 1920s, there were only a few major manufacturers left and "You can have any color you want as long as it's black" dominated. It took some forty years for "consumerism" to become an important market force in the automobile world.

Perhaps active consumerism on the part of people who design paper and other machines will be more effective in the computer area. Possibly the knowledgeable current users can clarify their demands and make them felt. Maybe even the masses of potential ordinary users will realize what computers could mean to themselves and press for the commercial availability of such computer services. But if all these fail, assuming the automobile analogy is valid, current twenty year predictions will finally be precipitated some forty years from now by some bright, young lawyer with a book entitled "Non Cost Effective at Any Speed."

REFERENCES

1 R BRENT D J KUCK K MARUYAMA
Parallel evaluation of arithmetic expressions without division
Submitted for publication

2 P BUDNIK D KUCK
The organization and use of parallel memories
IEEE Trans Comput Vol C-20 December 1971 pp 1566-1569

3 F T COYLE
The hidden speed of ISAM
Datamation June 1971

4 E W DAVIS JR
A multiprocessor for simulation applications
PhD Thesis University of Illinois at Urbana-Champaign
Department of Computer Science Report No 527 June 1972

5 E W DAVIS JR
Concurrent processing of conditional jump trees
Compcon 72 IEEE Computer Society Conference
Proceedings September 1972 San Francisco

6 B DE LUGISH
A class of algorithms for automatic evaluation of certain elementary functions in a binary computer
PhD Thesis University of Illinois at Urbana-Champaign
Department of Computer Science Report No 399 June 1970

7 P J DENNING
Virtual memory
Computing Surveys Vol 2 No 3 September 1970

8 D E GOLD
Elimination of rotational latency by dynamic disk allocation
PhD Thesis University of Illinois at Urbana-Champaign
Department of Computer Science Report No 522 May 1972

9 D E GOLD
Applications of some switching network results to dynamic allocation of memories in a hierarchy
Compcon 72 IEEE Computer Society Conference
Proceedings September 1972 San Francisco

10 P W KRASKA
Parallelism exploitation and scheduling
PhD Thesis University of Illinois at Urbana-Champaign
Department of Computer Science Report No 518 June 1972

11 D KUCK
Student memo
Unpublished University of Illinois at Urbana-Champaign
Department of Computer Science December 1971

12 D J KUCK D H LAWRIE
The use and performance of memory hierarchies—A survey
Software Engineering Vol 1 Academic Press Inc New York and London 1970

13 D J KUCK D H LAWRIE Y MURAOKA
Interconnection networks for processors and memories in large systems
Compcon 72 IEEE Computer Society Conference
Proceedings September 1972 San Francisco

14 D J KUCK Y MURAOKA
Fast computers from slow parts

Compcon 72 IEEE Computer Society Conference
September 1972 San Francisco

15 D J KUCK Y MURAOKA S C CHEN
On the number of operations simultaneously executable in Fortran-like programs and their resulting speed-up
To be published in IEEE Trans Computers

16 D H LAWRIE
On the design of disc processors
Unpublished Memo 1969

17 E MORENOFF W BECKETT P G KESEL
F J WINNINGHOFF P M WOLFF
4-way parallel processor partition of an atmospheric primitive-equation prediction model
Proceedings of AFIPS Spring Joint Computer Conference 1971 AFIPS Press

18 Y MURAOKA
Parallelism exposure and exploitation in programs
PhD Thesis University of Illinois at Urbana-Champaign Department of Computer Science Report No 424 February 1971

19 E J POLLEY JR
An assembler for efficient file manipulation
MS Thesis University of Illinois at Urbana-Champaign Department of Computer Science Report No 534 August 1972

20 J A RUDOLPH L C FULMER
W C MEILANDER
With associative memory, speed limit is no barrier
Electronics June 1970

21 W F SHARPE
The economics of computers
Columbia University Press New York 1969

22 H YAMADA
Emulation of disc file processor
MS Thesis University of Illinois at Urbana-Champaign Department of Computer Science Report No 436 June 1971

The TI ASC—A highly modular and flexible super computer architecture*

by W. J. WATSON

Texas Instruments Incorporated
Dallas, Texas

INTRODUCTION

Early in 1966, a large computer development program was begun by Texas Instruments. The goal for this effort was to provide needed capacity for supporting seismic processing, plus offering a general super computer capability in the support of new markets.

This development has resulted in the Advanced Scientific Computer (ASC)—a highly modular system offering a wide spectrum of computing power and configurability.

OVERVIEW OF THE SYSTEM

The major subsystems of a typical configuration are shown in Figure 1: the central memory, the central processor, the peripheral processor, on-line bulk storage, a digital communications interface, plus a selection of standard peripherals.

The peripheral processor has been designed for executing the operating system. The central processor has been designed expressly to provide high computing power for large arrays of data. The central processor operates as a slave to the peripheral processor. This design approach was chosen to maximize the overlapping of system overhead tasks with the execution of user programs. In operation the job stream is analyzed by the peripheral processor. The language processors, plus user object code, are executed by the central processor. System control and I/O tasks are processed by the peripheral processor. I/O is routed through high-speed, head-per-track disc storage. A data communications interface for the common carriers is provided for the support of remote batch and interactive terminals. Standard types of peripherals are also provided. The central memory serves as the common access communications and access storage medium for these subsystems.

CENTRAL MEMORY

The ASC central memory consists of a memory control unit (MCU) and appropriately sized modules of high-speed or medium-speed central memory. Optionally, a medium-speed central memory extension can be used in conjunction with a high-speed memory.

The MCU is organized as a two-way, 256-bit/channel (8-word) parallel access traffic net between eight independent processor ports and nine memory buses, with each processor port having full accessibility to all memories. The nine memory buses are organized to provide eight-way interleaving for the first eight buses with the ninth bus used for the central memory extension. The MCU provides the facilities for controlling access from the eight processor ports to a CM having a 24-bit address space (16 million words). A port expander can be utilized to expand the number of processor ports. Figure 2 illustrates this structure.

The MCU is designed to operate asynchronously, independent of cable delays, processor clock rates, and memory unit access and cycle times. This capability allows for a great deal of flexibility to accommodate improvements in memory or processor technologies which may be desired. The MCU is capable of handling a maximum data transfer rate of 80 million words per second per port, giving a total transfer capacity of 640M words per second. Therefore, a significant capacity beyond today's memory and processor speeds is available in the MCU.

The semiconductor high-speed central memory modules have a cycle time of 160 ns and a read time of 140 ns. Additionally, all transfers are 256 bits

* A description of the Advanced Scientific Computer System, by the Staff of Texas Instruments (Internal Texas Instruments Publication M1001P).

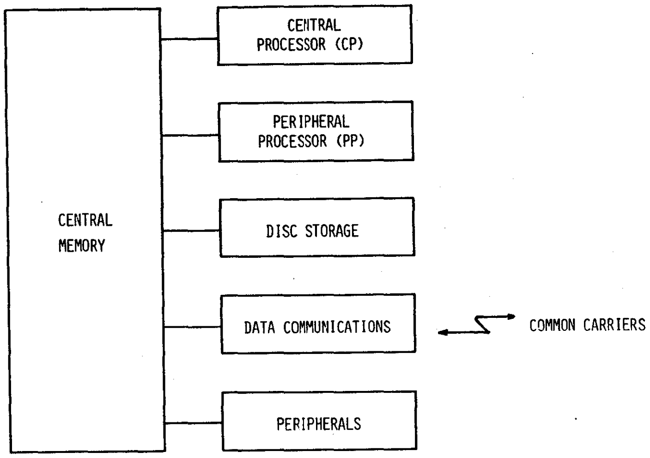


Figure 1—Major ASC subsystems

(eight 32-bit words) with a Hamming code providing single-bit error correction and double-bit error detection for each 32-bit word. High-speed central memory is typically divided into eight equal sized modules which permits eight-way interleaving. A patch board within the MCU controls the memory address decoding and sets the interleaving pattern.

The optional central memory extension provides for large amounts of relatively economical medium-speed memory to be utilized in support of the high-speed central memory. The memory extension uses 1 μs semiconductor technology and is also accessed in 8-word increments. Single-bit error correction is provided at the 8-word level. The central memory extension is included in the address space of the central memory and, therefore, can be addressed by a processor or channel controller for instructions or operands. It is also possible to effect block transfers of data between high-speed memory and the memory extension. This is possible because both a normal memory bus and a memory access port are provided. Block transfers are initiated by the peripheral processor with the specification of the source starting address, the destination starting address, and the block length. The block transfer proceeds automatically at 40M words per second, and the peripheral processor is notified upon completion.

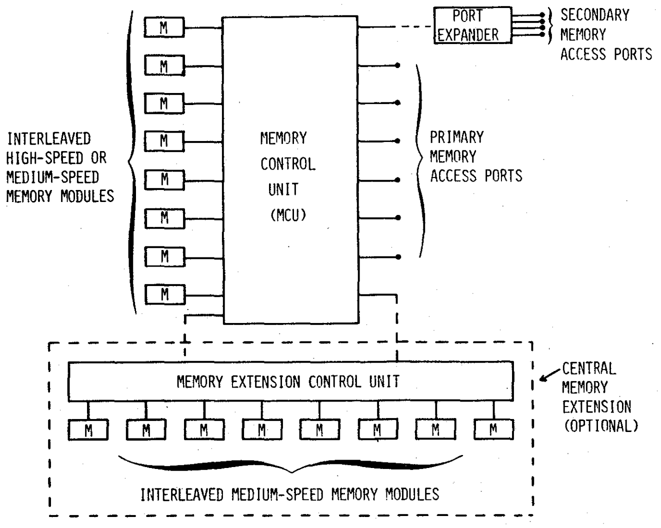
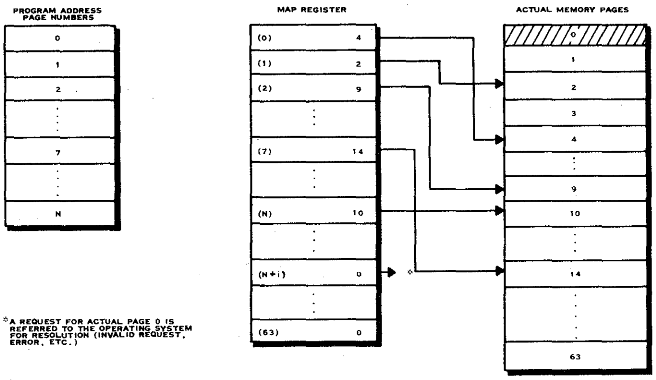

INTERLEAVED HIGH-SPEED OR MEDIUM-SPEED MEMORY MODULES
M
MEMORY CONTROL UNIT (MCU)
PORT EXPANDER
SECONDARY MEMORY ACCESS PORTS
PRIMARY MEMORY ACCESS PORTS
MEMORY EXTENSION CONTROL UNIT
CENTRAL MEMORY EXTENSION (OPTIONAL)
INTERLEAVED MEDIUM-SPEED MEMORY MODULES

Figure 2—Modular structure of the ASC central memory

The central memory size is limited only by the 24-bit address (16 megawords). The proportions of fast memory and memory extension may be varied in order to balance memory capacities to suit the particular system requirements. The present high-speed memory module is modular from 16K to 128K 32-bit words. This permits memories from 128K to one million words to be configured.



*A REQUEST FOR ACTUAL PAGE 0 IS REFERRED TO THE OPERATING SYSTEM FOR RESOLUTION (INVALID REQUEST, ERROR, ETC.)

Figure 3—Memory mapping

Central memory management and access control of memory ports is achieved through the use of two facilities: map registers and protect registers. Each user program has its own unique page address map. Page addresses not required by the program are mapped into absolute page zero which is not accessible to the CP. When a program is loaded into memory, it will likely be loaded into discontiguous memory pages. During program execution, program developed page addresses are converted, without execution time penalty, to actual page addresses by the map registers. Because a reference to page zero is denied and the relevant processor notified, the map registers provide for inter-user memory protection. Figure 3 shows the mapping scheme. Desired page sizes depend on the amount of central memory and the problem mix of a particular installation. Four different page sizes may be specified for an ASC system, varying from 4K to

256K words. A program may utilize any one of the page sizes available.

The protect registers allow for intra-user protection. These registers consist of three pairs of bounds registers for defining the upper and lower addresses of access for read, write, or execute areas. The five combinations of protection presently used by the system software with the bounds registers are:

- Execute Only
- Read Only
- Execute, Read, No Write
- Read, Write, No Execute
- Read, Write, Execute

An attempt to reference an area out of bounds for a particular control state is denied and the processor notified of the attempted violation.

In large ASC systems, more processors and control units require additional access ports to memory. In these cases memory port expanders are utilized to provide additional ports and are utilized to service the devices not requiring the full bandwidth of a memory port. Each memory access port expander provides a 1:4 expansion with a maximum bandwidth degradation of ten percent; i.e., from 80 million 32-bit words per second to approximately 72 million 32-bit words per second. These expanders can be concatenated to provide further increases in connectivity. Priorities at the single access port interface are resolved on either a fixed or distributed basis. The mode is selected by patch card wiring in the expander hardware.

CENTRAL PROCESSOR

The central processor (CP) provides both scalar (single operand) and vector (array) instructions at the machine level. The basic instruction size is 32 bits, with 16-, 32-, or 64-bit operands. The single instruction stream, which contains a mixture of scalar and vector instructions, is preprocessed by the instruction processing unit.

The central processor design is such that one, two, three, or four execution units or "pipes" can be provided. These units employ the pipeline concept in both scalar and vector modes. A single execution unit can have up to twelve scalar instructions in process at one time. From one to four vector results can be produced every 60 ns, depending on the number of execution units provided.

The CP has 48 program-addressable registers. This group of 32-bit registers consists of sixteen base address registers, sixteen arithmetic registers, eight index registers, and eight vector parameter registers. This last group is used to extend the instruction format for the complete specification of vector instructions. The basic instruction format is shown as it relates to these register groups in Figure 4.

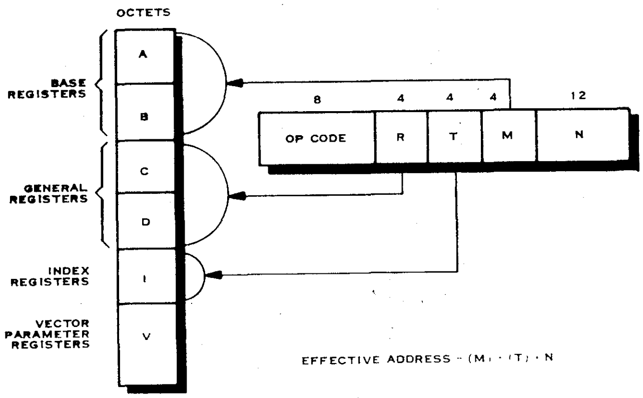


Figure 4—Instruction format and register groups

The CP scalar instruction repertoire includes an extensive set of Load and Store instructions: halfword, fullword, and doubleword instructions, with immediate, magnitude, and negative operand capabilities. Ability to load and store register files and to load effective addresses is also available. Arithmetic scalars include various adds, subtract, multiply, and divide for halfword (16-bit) and fullword (32-bit) fixed point numbers and fullword and doubleword (64-bit) floating point numbers. Scalar logical instructions are provided as are arithmetic, logical, and circular shifts. Various comparison instructions and combination comparison-logical instructions are provided for halfword, fullword, and doublewords. Many combinations of test and branching instructions with incrementing or decrementing capability are also available. Stacking and modifying arithmetic registers can be done with single instructions. Subroutine linkage is accomplished through Branch and Load instructions. Format conversion for single and doublewords, as well as normalize instructions, are available.

The vector capabilities of the CP are made available through the use of VECTL (vector after loading vector parameter file) and VECT (assumes parameter file is already loaded) instructions. The vector repertoire includes such arithmetic operations as add, subtract, multiply, divide, vector dot product, matrix multiplication, and others for both fixed point and floating point representations. Vector instructions are also available for shifting; logical operations; comparisons; format conversions; normalization; and special operations—such as Merge, Order, Search, Peak Pick, Select and Replace, among others.

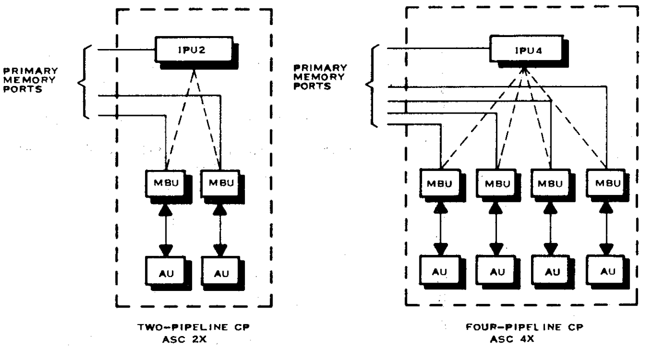


Figure 5—Basic structure of the CP

One important characteristic of the vector instruction capability is the ability to encompass three dimensions of addressability within a single vector instruction. This is equivalent to a nest of three indexing loops in a conventional machine.

The basic structure of the CP, shown in Figure 5, has three major components: the instruction processing unit (IPU) for non-arithmetic stages of instruction processing for the CP instruction stream, the memory buffer unit (MBU) to provide operand interfacing with the central memory, and an arithmetic unit (AU) to perform the specified arithmetic or logical operations. Figure 5 shows a CP diagram for 2- or 4-pipeline CP's, each with a corresponding number of MBU-AU pairs. Note that a memory port is required for the IPU and, in addition, one memory port for each pipeline (MBU-AU pair) in a CP.

A significant feature of the CP hardware is an operand look-ahead capability which causes memory references to be requested prior to the time of actual need. Double buffering in multiple 8-word (octet) buffers for each pipeline provides a smooth data flow to and from each arithmetic unit. The pipelined AU achieves its highest sustained flow rate in the vector mode, typically a result each 60 ns per AU.

Instruction processing unit

The primary function of the instruction processing unit (IPU) is to supply a continuous stream of instructions for execution by the other parts of the CP. One Central Memory port is required to provide the instruction stream. Two 8-word (octet) buffers are utilized to achieve a balanced stream of instructions from memory to the IPU. Instructions are transferred from memory in octets as are all other references to memory for fetching or storing of information.

The following functions are performed by the IPU: (1) instruction fetch, (2) instruction decode, (3) register operand selection, (4) effective address development through indexing and/or indirect addressing, (5) immediate operand development, (6) branch address development, (7) determination of branch condition, (8) storage of AU results into the register file, (9) scalar hazard and register conflict resolution, (10) generation of vector starting addresses, and (11) transmittal of vector parameters to the MBU during vector initialization.

Up to 36 instructions in various stages of execution can be overlapped within the 4-pipe CP. There are twenty positions for instructions in the 2-pipe CP and twelve positions for instructions in the 1-pipe CP. Four levels are contained within the IPU, and eight levels are contained in each arithmetic pipeline (MBU-AU pair). In addition to the previously mentioned functions, the IPU performs routing of instructions to the MBU-AU pairs based on an optimum use of arithmetic unit capability.

Vector processing is altered by software in order to distribute segments of the vector for multiple pipe systems.

Several features are provided to alleviate the potential problems of branches and instruction dependencies in the instruction pipeline. The Prepare-to-Branch instruction, used extensively by the Fortran compiler, increases the execution speed of branches, particularly important in loop iterations. This instruction provides the IPU control hardware with advance address information to facilitate uninterrupted instruction processing. Instruction dependencies are recognized by the hardware. It scans the instruction stream and distributes the independent instructions across MBU-AU pairs to insure proper, yet efficient, execution sequences.

Memory buffer unit

The memory buffer unit (MBU) provides an interface between central memory and the arithmetic unit. Its primary function is to supply the arithmetic unit with a continuous stream of operands from memory and to provide for the storing of the results back to memory. Note that all references to memory, whether for fetching or storing, are made in 8-word increments (octets).

The MBU has three double buffers, one octet per buffer, called the "X" and "Y" buffers for input and the "Z" buffers for output. This double buffering is provided so that pipeline processing can be sustained at a high rate with minimal memory access conflicts These buffers are illustrated in Figure 6.

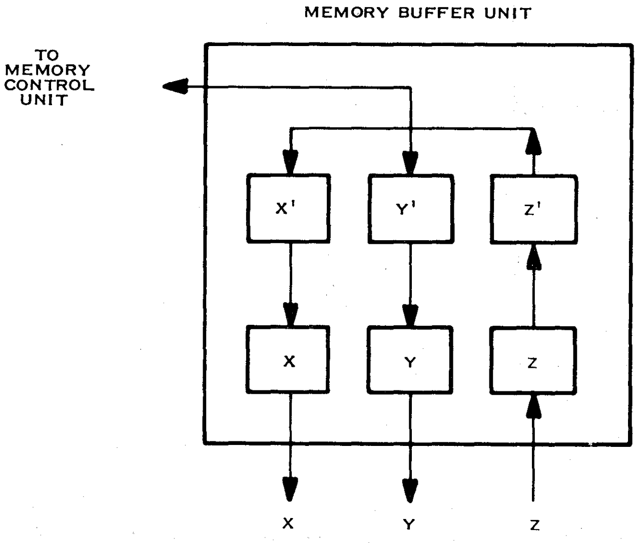


Figure 6—Multiple operand streams in the memory buffer unit

Dur'ng scalar operations, data specified by effective addresses developed in the IPU are fetched or stored as required. The Z buffer can be transferred directly to the X or Y buffers so that memory references are not necessary for scalar operands which res'de in the Z buffer.

For most vector operations, two operand data strings are fetched, while a result data string is stored. Addresses for sustaining the vector operations are computed in the MBU using parameters initially specified by the vector parameter file.

Arithmetic unit

The primary function of a CP arithmetic unit (AU) is to perform the arithmetic operations specified by the operation code of the instruction currently at the AU level. There is one AU per pipeline in the CP, each having a 60 ns basic cycle time. A distinguishing feature of an AU is the pipeline structure which allows efficient execution of the arithmetic part of all instructions. There are eight exclusive partitions of the AU pipeline involved, each of which can provide an output every 60 ns. These e'ght sections are (1) received register, (2) exponent subtract, (3) align, (4) add, (5) normalize, (6) multiply, (7) accumulate, and (8) output. Figure 7 shows how different sections of the AU are utilized for execution of particular instructions; i.e., floating point addition and fixed point multiplication.

An AU is a 64-bit parallel operating unit for most scalar and vector instructions. Exceptions are double length multiply and all types of division. In these circumstances various combinations of the components of the AU are utilized; and, therefore, more than one clock cycle is required to complete these arithmetic operations.

Fixed point negative numbers are represented in

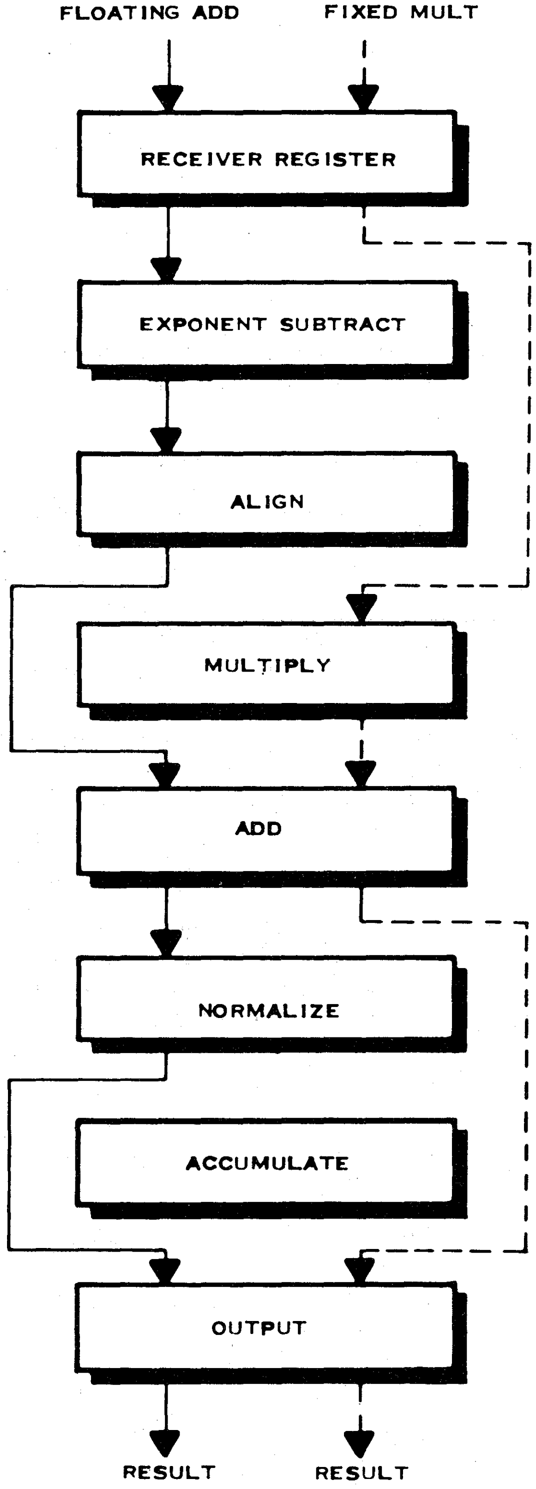


Figure 7—Arithmetic unit pipeline concept

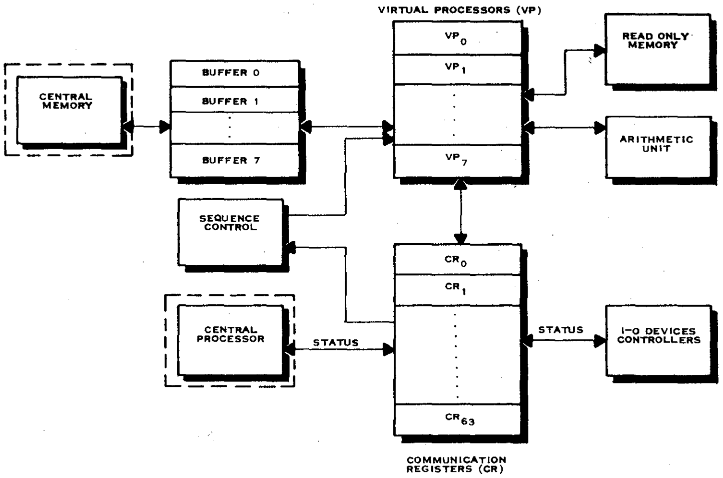


Figure 8—Peripheral processor

two's complement notation, and the floating point representation is hexadecimal with the exponent biased by $40_{(16)}$.

THE PERIPHERAL PROCESSOR

The peripheral processor (PP) is a powerful multiprocessor designed to perform the control and data management functions of the ASC. Several aspects of the implementation of the peripheral processor concept greatly increase the effectiveness of the ASC system. Figure 8 shows the logical organization of the PP.

The PP is a collection of eight individual processors called virtual processors (VP's). Each VP has its own program counter along with arithmetic, index, base, and instruction registers. The eight VP's share a read only memory, an arithmetic unit, an instruction processing unit, and a central memory buffer. Use of the common units is distributed among the VP's using sixteen single 85 ns cycles. When an equally distributed sequence of time units is used, each of the eight VP's receives two 85 ns cycles every 1.4 μs. The typical PP instruction requires two 85 ns cycles for completion.

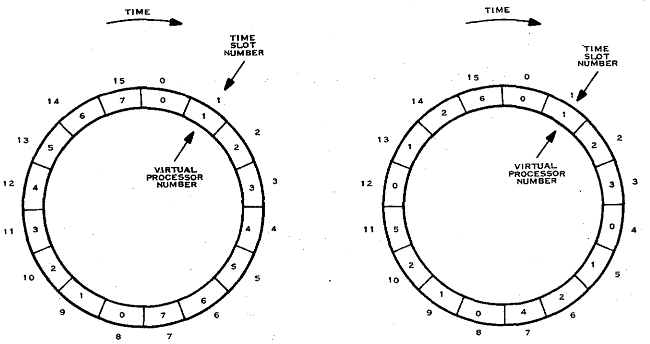


Figure 9—Two possible VP time slot assignments

The distribution of available time units can be dynamically varied to suit particular processing requirements. Figure 9 illustrates two possible distributions.

The read only memory within the PP is utilized for program storage and execution of those short routines which are highly utilized by the VP's, such as polling loops. The read only memory consists of up to 4K 32-bit words of non-volatile memory elements with a cycle time of less than 85 ns. It is modular in 256-word increments.

Because the PP is intended to perform control functions rather than execute mathematical algorithms, the instruction set is oriented toward control operations and does not require multiplication, division, or floating point operations. The instruction format is similar to that of the central processor, using a 32-bit word for each instruction. Instructions are provided for bit (1 bit), byte (8 bits), halfword (16 bits), and fullword (32 bits) operations.

Each VP has direct access to the entire central memory for program execution and data storage. Therefore, a single copy of reentrant code can be executed simultaneously by more than one VP.

The communications register (CR) file contains sixty-four 32-bit word registers which are program addressable by the VP's. The CR file serves as the principal storage media for control information necessary for the coordination of all parts of the ASC system. Synchronization of communications is achieved between all processors (CP, VP's, channel controllers, and peripheral unit controllers) from interpretat on of status bits received from all devices into the CR file.

DISC STORAGE

Disc storage is the principal secondary storage system for the ASC system. Disc storage consists of head-per-track (H/T) disc systems supplemented by positioning-arm disc (PAD) systems.

Head-per-track (H/) disc system

The H/T disc system is a high-performance device whose effective performance is further enhanced because the operating system utilizes a shortest-access-time-first (SATF) algorithm[1] for data transfers. This combination of hardware and software provides a very high effective transfer rate. Each H/T disc module has a capacity of 25 million 32-bit words with a transfer rate of approximately 500K words per second. Using the shortest-access-time-first algorithm, access time will average approximately 5 ms which results in an exceptionally fast "effective" transfer rate. The rotational period of the disc is 32 ms. Each H/T disc module has seven discs with fourteen surfaces. Two surfaces of the module are used as alternate storage for inoperative sections. For data ordering purposes, the discs are divided into bands and then further subdivided into sectors of 64 words each.

Positioning-arm disc (PAD) system

The PAD system, when utilized to supplement head per track, is available in a variety of configurations. Control of PAD systems is achieved by use of channel interface, disc controller, and disc interface units. From two to eight PAD disc drives may be attached to a set of control devices. The number of controllers and discs per controller will depend upon the storage and retrieval problem requirements.

The PAD system has a transfer rate of 200K words per second and a storage capacity of 25M words per disc drive. Access time is divided into two categories: positioning-arm time which is 30 ms average with a maximum of 55 ms and average rotational latency which is 8.4 ms. Thus, average total access time is approximately 38 ms.

DATA COMMUNICATIONS

The data communication system is very modular and, thus, externally flexible in the various devices which may be utilized for communication with the ASC. Data communications are controlled by a data concentrator which, in turn, interfaces to the MCU through a channel control device.

Data concentrator

The data concentrator is a TI-980 minicomputer equipped with special-purpose hardware communication interface units on its direct memory access ports. The TI-980 is a small, general-purpose computer with up to 64K 16-bit words of memory and a one-microsecond cycle time. The data concentrator hardware is under control of a data communications operating system which executes in the TI-980. This operating system provides for the functions of buffering, reformatting, routing, protocol handling, error control and recovery procedures, and system control messages. The system services multiple stations concurrently.

The data communications system presently supports communication with three types of stations: high-performance user terminals, other large computers, and remote concentrators. The system can be easily extended to support smaller terminals down to the teletype level. These stations may be either remote or local. When local, the communication link is implemented with multiple conductor cables. Since the transfer is asynchronous by word, the average transfer rate is very dependent upon cable length with a maximum transfer rate of 250,000 words per second for distances less than 500 feet.

Remote links

Remote links are presently implemented with non-switched, full duplex common carrier data transmission facilities. Data is transferred over these links synchronously at rates determined by the modems and common carrier bandwidths. The data communication system supports transfer rates up to a maximum of 240,000 bits per second. Because the system supports full duplex transmission, this capacity typically translates to the ability to support a 1200 lpm printer simultaneously with a 1000 cpm reader over a 9600 bps transmission facility.

PERIPHERALS

Standard types of magnetic tape drives, card equipment, and printers have been interfaced with the ASC. These interfaces are attached to primary or secondary memory ports through a variety of standard selected and multiplexed data channels.

SUMMARY

Preservation of global system modularity concepts in the design of the ASC has resulted in a capability for configuring systems having a very wide range of cost and capabilities.

In the memory area capacity, performance, connectivity, protection, and mapping are all variable over wide bounds. The central processor can be tailored to provide a wide range of processing power by using one, two, three, or four pipes.

The peripheral processor provides for dynamically matching the execution rates of up to eight independent instruction streams with the task requirements. The

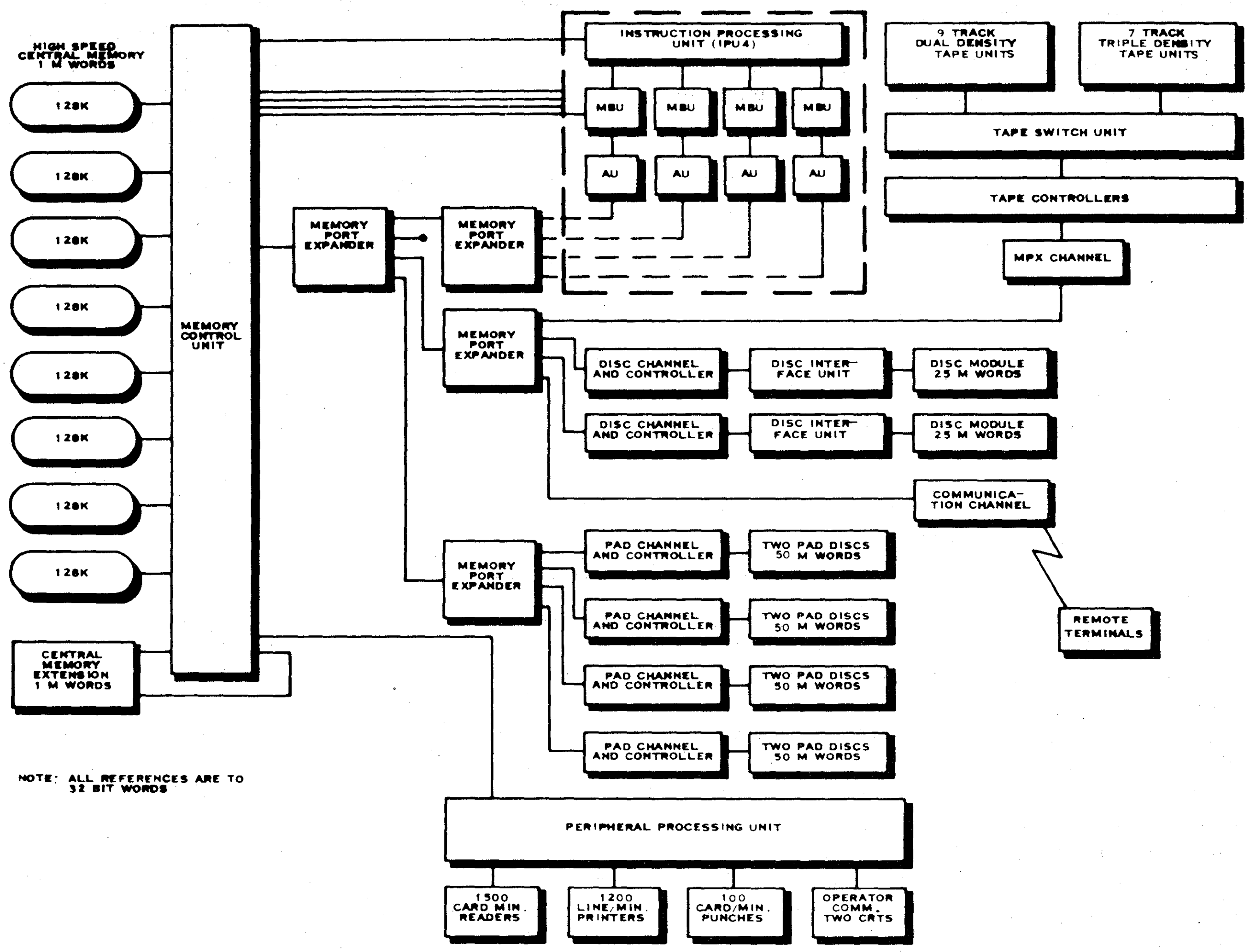


Figure 10—A possible ASC system configuration

highly flexible communication register file provides a matrix of 2048 bits which can be manipulated and sensed by the eight virtual processors. Flexible hardware interfaces are provided for coupling these bits to external I/O signal lines. Finally, the modular read only program memory of the peripheral processor accommodates growth and modifications in read only memory resident operating system code.

An example of a complete system configuration is illustrated in Figure 10.

ACKNOWLEDGMENTS

Although it would not be possible to acknowledge all of the contributors to the ASC program, particular recognition should be given to Messrs. H. G. Cragon, W. D. Kastner, E. H. Husband, D. R. Best, C. M. Stephenson, C. R. Hall, F. A. Galindo, and E. C. Garth, all of whom contributed immeasurably to the architecture of the ASC system. Many other members of the Texas Instruments Equipment Group staff have also made significant contributions in the development of the ASC system.

REFERENCE

1 P J DENNING
Effects of scheduling on file memory operations
Proceedings of the Spring Joint Computer Conference
1967

A production implementation of an associative array processor—STARAN

by JACK A. RUDOLPH

Goodyear Aerospace Corporation
Akron, Ohio

INTRODUCTION

The associative or content-addressed memory has been an attractive concept to computer designers ever since Slade and McMahon's 1957 paper[1] described a "catalog" memory. Associative memories offered relief from the continuing problem presented by the typical coordinate-addressed memory which requires that an "address" be obtained or calculated before data stored at that address may be retrieved. The associative memory could acquire in a single memory access any data from memory without pre-knowledge of its location. Ordered files and sorting operations could be eliminated. Unfortunately, early associative memories were expensive, hence none found their way as the "main frame" memory into any commercial computer design.

The organization of an associative memory (AM) requires that each n-bit physical word of the memory be connected to a dedicated processing element (PE) which performs the compare function between a bit read non-destructively from the word and a corresponding input bit from a query word. The PE's for all words are driven by a central controller, thus a single query bit is simultaneously compared with the corresponding stored bit in every word of the AM. With the ability to simultaneously write back the state of each PE into a specified bit position of each word it became possible to perform bit-serial arithmetic between fields of bits within each physical memory word. An array of associative memory words could then be viewed as an array of simple computers—an associative array processor—with all the simple computers in the array simultaneously executing the same instruction obtained from a common control unit as is done in the more complex ILLIAC-IV design.

An alternative AP design provides a PE at each bit of each physical memory word. This design, though complex in terms of logic and interconnection requirements, permits a simultaneous compare of all bits in a query word with all bits of the memory word rather than the serial-by-bit operation described earlier.

Due to the early high cost of semi-conductor memory and logic elements none of the many associative processor designs described in the literature were attractive enough to warrant development. However, it has now become commercially feasible to construct a computing system embodying "main frame" memory content addressability coupled with array arithmetic capability operating under a more or less conventional stored program control system.

Several proprietary versions of the associative processor (AP) are being developed. The first working engineering model[2] known to the author, built for USAF by Goodyear Aerospace Corporation, was demonstrated during a Tri-Service contract review in June, 1969 at Akron, Ohio. The same machine, modified to include a larger instruction memory, was loaned[3] by USAF in 1971 to the FAA for conflict detection tests in a live air traffic control terminal environment at Knoxville, Tennessee operating in a multi-computer configuration with a Univac 1230 conventional computer. The original test objectives were achieved by December, 1971 and additional experiments involving terrain avoidance processing were completed successfully in June, 1972.

The lessons learned in programming and testing the USAF AP model resulted in a new design called STARAN S which was committed to production in 1971. This first commercial AP was publicly introduced in a series of live demonstrations in May, 1972 at the TRANSPO exhibit in Washington, D.C. and in June, 1972 at Boston, Mass.

This paper describes STARAN S and its programming language, provides examples of its applications, and discusses measures of AP cost-effectiveness.

STARAN* DESCRIPTION

A configuration diagram of STARAN S is shown in Figure 1. Studies have shown that initial uses of AP's would be weighted toward real-time applications involving interface with a wide variety of sensors, conventional computers, signal processors, interactive displays, and mass storage devices. To accommodate all such interfaces the STARAN system was divided into a standardized main frame design and a custom interface unit. A variety of I/O options implemented in the custom interface unit include conventional direct memory access (DMA), buffered I/O (BIO) channels, external function channels (EXF) and a unique interface called parallel I/O (PIO).

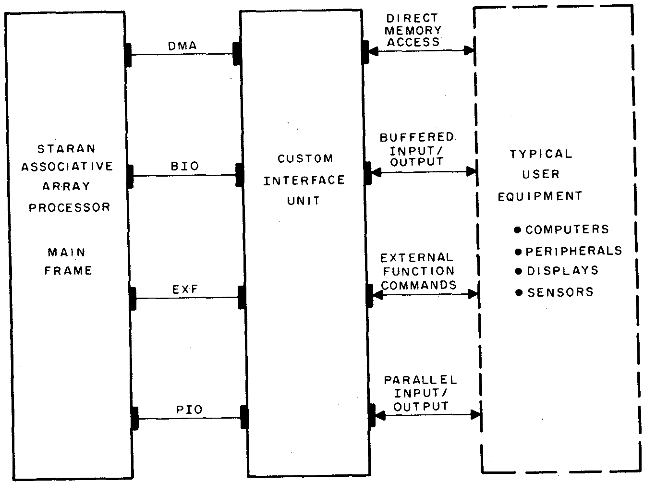


Figure 1—STARAN system configuration

A top-cut diagram of the STARAN main frame is shown in Figure 2. It consists of a conventionally addressed control memory for program storage and data buffering, a control logic unit for sequencing and decoding instructions from control memory and from one to thirty-two modular AP arrays.

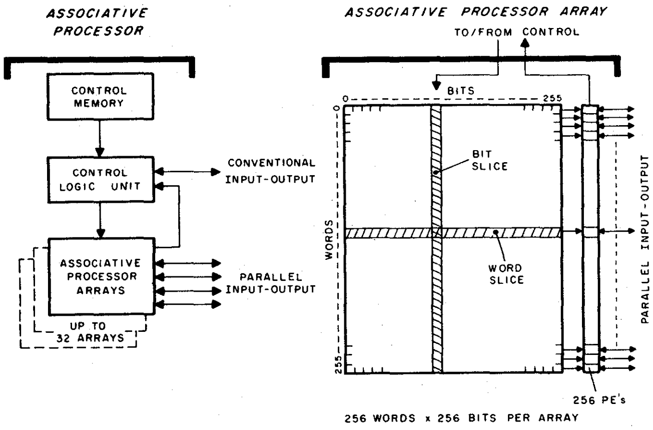


Figure 2—Associative processor diagrams

A typical AP array is also shown in Figure 2. This key element of the STARAN S computer system is the "main frame" memory which provides content addressability and parallel processing capabilities. Each array consists of 65,536 bits organized as a multi-dimensional access memory matrix of 256 words by 256 bits with parallel access to up to 256 bits at a time in either the word or bit direction. In addition to the storage elements, each array contains 256 bit-serial PE's often referred to in associative memory literature as the response store. The unique PIO capability is provided by the response store, where every PE has an independent external device I/O path. Control signals generated by the control logic unit are fed to the processing elements in parallel and all processing elements execute the instruction simultaneously. As additional arrays are added to the system these are also connected in parallel to the control logic unit, thus application programs need not be modified as the capacity of the system increases.

Major elements of the STARAN block diagram shown in Figure 3 are described below:

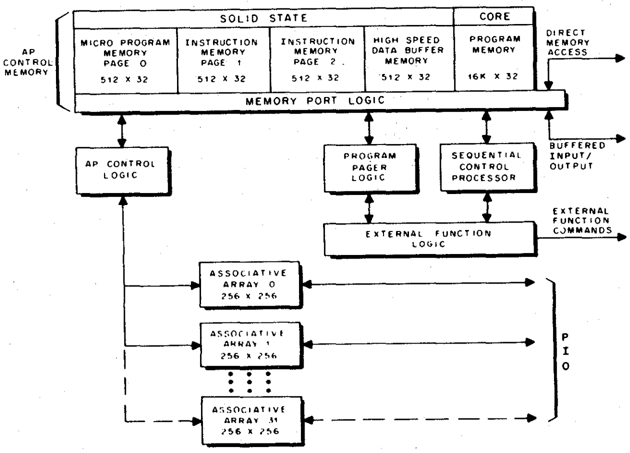


Figure 3—STARAN basic block diagram

* T. M. Goodyear Aerospace Corporation, Akron, Ohio

AP control memory

The conventionally addressed and indexed AP control memory is used to store assembled AP application programs. It is also used for data storage and to act as a buffer between AP control and other elements of STARAN S. The AP control memory and associative array cycles are overlapped.

Control memory is divided into several memory blocks. Three fast "page" memories contain the current AP program segments; the slower core memory contains the remainder of the AP program. A program pager transfers program segments from the slow to the fast memory blocks. Control memory words contain 32 bits of either data or instructions.

The "page" memories use volatile, bipolar, semiconductor elements. A page contains 512 words but can be doubled to 1024 words each on an optional basis. Page 0 may contain a library of microprograms such as arithmetic subroutines. Pages 1 and 2 are used in ping-pong fashion, with AP control executing instructions out of one page while the other is being loaded by the program pager. This permits use of the page memories for selected segments of the program or for the entire program if fast execution is required.

The high-speed data buffer (HSDB), like the page memories, uses volatile, bipolar, semi-conductor elements. It contains 512 words but also can be doubled to 1024 words. All buses can access the HSDB to store data or instruction items that need to be accessed quickly by the different STARAN elements.

The bulk core memory uses nonvolatile core storage. It contains 16,384 words and is optionally expandable to 32,768 words. It is used for storing complete AP application programs. Since the bulk core memory is accessible to all buses it is useful as a buffer for data items that do not require the high-speed of the HSDB.

A block of up to 30,720 AP control memory addresses is reserved for the direct memory access (DMA) channel to external memory. All buses can access the DMA block, thus it is possible to operate the AP solely from programs stored on external memory as, for example, the main frame memory of a conventional computer.

AP control logic

Executing instructions from control memory, AP control logic directly manipulates data within the associative arrays and is the data communication path between control memory and the arrays.

Program pager logic

The program pager loads the fast page memories from the slow core memory. While the AP control is executing a program segment out of one page, the pager can be loading the other page with a future program segment.

External function logic

External function (EXF) logic enables the AP control, sequential control, or an external device to control the STARAN S operation. By issuing external function codes to EXF a STARAN S element can interrogate and control the status of the other elements.

Sequential control processor

The sequential control (SC) portion of STARAN S consists of a sequential processor having an 8K 16-bit memory, a keyboard-printer, a perforated tape reader/punch unit, and logic capability to interface the sequential processor with other STARAN S elements. SC is used for system software programs such as assembler, operating system, diagnostic programs, debugging, and housekeeping routines. SC peripherals which may be useful programming aids are available as options.

Input/output options

A custom interface unit (not shown in Figure 3) can provide any required combination of DMA, BIO, EXF, or PIO channels. A DMA channel to a conventional computer, for example, would permit rapid interchange of data between the systems in the common memory bank. The unique parallel I/O (PIO) channel, with a width of up to 256 bits per array, provides an extreme width channel up to 8192 bits wide at transfer rates in the sub-microsecond region. For example, a four-array STARAN S can input or output 1024-bit word or bit slices at an average slice rate exceeding 3 megacycles/sec providing an I/O bandwidth many times wider than that of a conventional computer. PIO provides a unique capability for large data base processing when used with wide bandwidth mass storage devices.

A photograph of a six array (model S-1500) STARAN is shown in Figure 4.

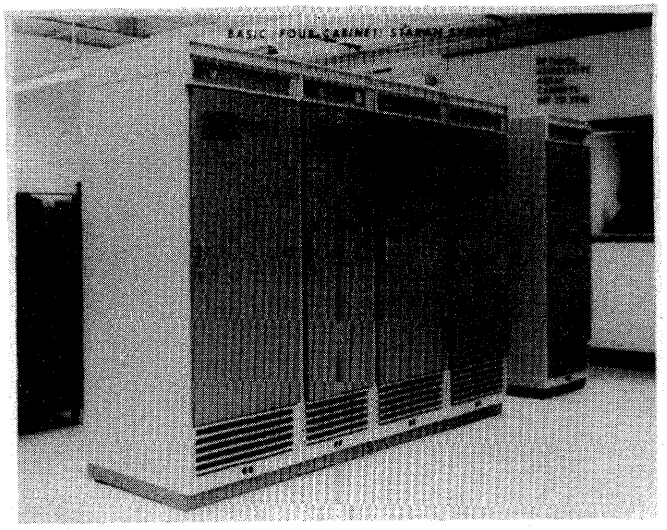


Figure 4—STARAN S-1500

ASSOCIATIVE PROCESSOR SOFTWARE

The STARAN software system consists of a symbolic assembler called APPLE (for **A**ssociative **P**rocessor **P**rogramming **L**anguag**E**), and a set of supervisor, utility, debug, diagnostic, and subroutine library program packages. An associative compiler has not yet been developed for STARAN. Early applications of STARAN must therefore be accomplished by assembly language programmers. Programmers find APPLE a convenient language to use, however, and write significantly fewer instructions to program a suitable application on STARAN than would have to be written for a conventional machine since APPLE's command structure reflects the content addressability and processing characteristics of the associative arrays the language controls. For example, although the programmer must explicitly define his record formats via field definition statements, he usually need not be concerned with physical record location in the arrays. Also, he need not order data tables by key, since any desired datum may be located in one parallel search operation. A third example of APPLE convenience is the elimination of the conventional programming loop which requires advancing a list pointer, examination of an exit criterion, and making a decision for each pass over different data sets. The APPLE array instruction processes all pertinent data sets simultaneously and does not require initialization of an index register with the count of data sets to be processed.

Internally, all software packages with the exception of array diagnostics and the subroutine library operate on the SC. In the minimum STARAN configuration the software packages are furnished on paper tape for input via the SC tape reader. Where STARAN is installed with interface to a conventional computer system in a multicomputer configuration, APPLE and supporting software can be input to STARAN using the existing peripherals of the conventional computer.

The usual load, store, test, branch, and control instructions required for sequential execution of an application program are present in APPLE. Where APPLE departs most from conventional assemblers is in the search and arithmetic array instructions. A representative set of fixed point standard instructions is shown in Table I with the approximate timing formulas. Hardware floating point is available on special order.

Associative search and arithmetic instructions are of two types, "argument register" and "field". In the first an operand (32 bits max) stored in the argument register of AP control is used as the search or arithmetic argument against a specified field in all array words simultaneously. Instructions of the field type perform similar operations but between specified fields within each array word.

Instruction execution times are dependent upon *n*, the number of bits in the operands (fields) involved in the instruction executions, but are not functions of the number of operands being processed, which relationship is exactly the opposite of that existing in the conventional computer. This characteristic dependence of execution time on operand or field length is a consequence of the word-parallel bit-serial design of the associative arrays discussed earlier.

From the programmer's point of view, Table I has interesting connotations; some of which are:

1. in real time applications the programmer can easily time out his initial flow diagram since programming loops in the conventional sense are eliminated. This single consequence of associative processing can save much of the reprogramming effort invariably found necessary during the testing phase of conventional attacks on real-time problems;
2. he can conserve on execution time (and array memory space) by defining fields to use only as many bits as are required by the application; and
3. he has no need for overhead-generating techniques such as indexed file constructions, linked lists, or sort and merge operations usually needed in a conventional computer. This capa-

TABLE I—Typical APPLE Associative Fixed Point Instructions

MNEMONIC	INSTRUCTION	APPROX. EXECUTION TIME (μs)**			MIPS* PER ARRAY FOR n=32
		FORMULA	n = 16	n=32	
ARGUMENT REGISTER INSTRUCTIONS					
EQC	EXACT MATCH COMPARAND	0.6+0.15n	3.0	5.4	47
GTC	GREATER THAN COMPARAND	0.7+0.15n	3.1	5.5	47
LTC	LESS THAN COMPARAND	0.7+0.15n	3.1	5.5	47
ADC	ADD AR TO FIELD	2.8+0.85n	16	30	8.5
FIELD INSTRUCTIONS					
EQF	EXACT MATCH FIELDS	0.6+0.43n	7.4	14	18
GTF	GREATER THAN FIELDS	2.3+0.43n	9.1	16	16
LTF	LESS THAN FIELDS	2.3+0.43n	9.1	16	16
MAXF	MAX FIELDS	0.6+0.68n	11	23	11
MINF	MIN FIELDS	0.6+0.68n	11	23	11
ADF	ADD FIELD TO FIELD	2.8+0.85n	16	30	8.5
MPF	MULTIPLY FIELD BY FIELD	5.8+2.9m+ 0.85mn+0.4	277	980	0.26

* Max execution rate of specified instructions for single array with all 256 PE's active.

** n or m equal number of bits in operand

bility results in a significant reduction both in the number of instructions which must be written and executed and the amount of memory required.

ARRAY STORAGE ALLOCATION

The concept of a file of related records as used in associative processing requires some discussion. In conventional approaches to file generation one thinks of the distinction between a logical file and a corresponding physical file; that is, a logical collection of records, usually ordered by some key, is placed as a block of contiguous addresses in a physical file. The conventional operating system keeps track of the beginning address and the block length for the file whether stored in core or on external stores. Thus in most cases logically different files are stored in physically separate areas of store.

The associative approach differs from the conventional approach in several ways: the records within the logical file need not and usually are not ordered by any key; records within a logical file usually are not stored in contiguous locations in an area of the array or on external devices; and the operating system generally is not required to keep track of individual file beginning addresses and block lengths.

In STARAN, records belonging to different logical files may be physically intermixed in the array as well as being logically unordered. Within each record format, in addition to defining the item fields, the programmer defines a set of control tag fields. How these tags are used is described below.

When new records are added to a logical file the update program writes the new, properly formatted record into the first available empty array location. Since empty array locations usually are not contiguously located within the array, records belonging to a specific file are scattered throughout the array in random locations. This characteristic is illustrated in the array map example of Figure 5.

Empty array memory locations are identified by executing an EQC on a one-bit activity tag field using an "O" as the search criteria. The execution time for this search (see Table I) is less than one microsecond at the end of which time all processing elements for physical memory words containing a 0 in the activity field will be in the "ON" state. At the conclusion of the search a hardware pointer automatically points to

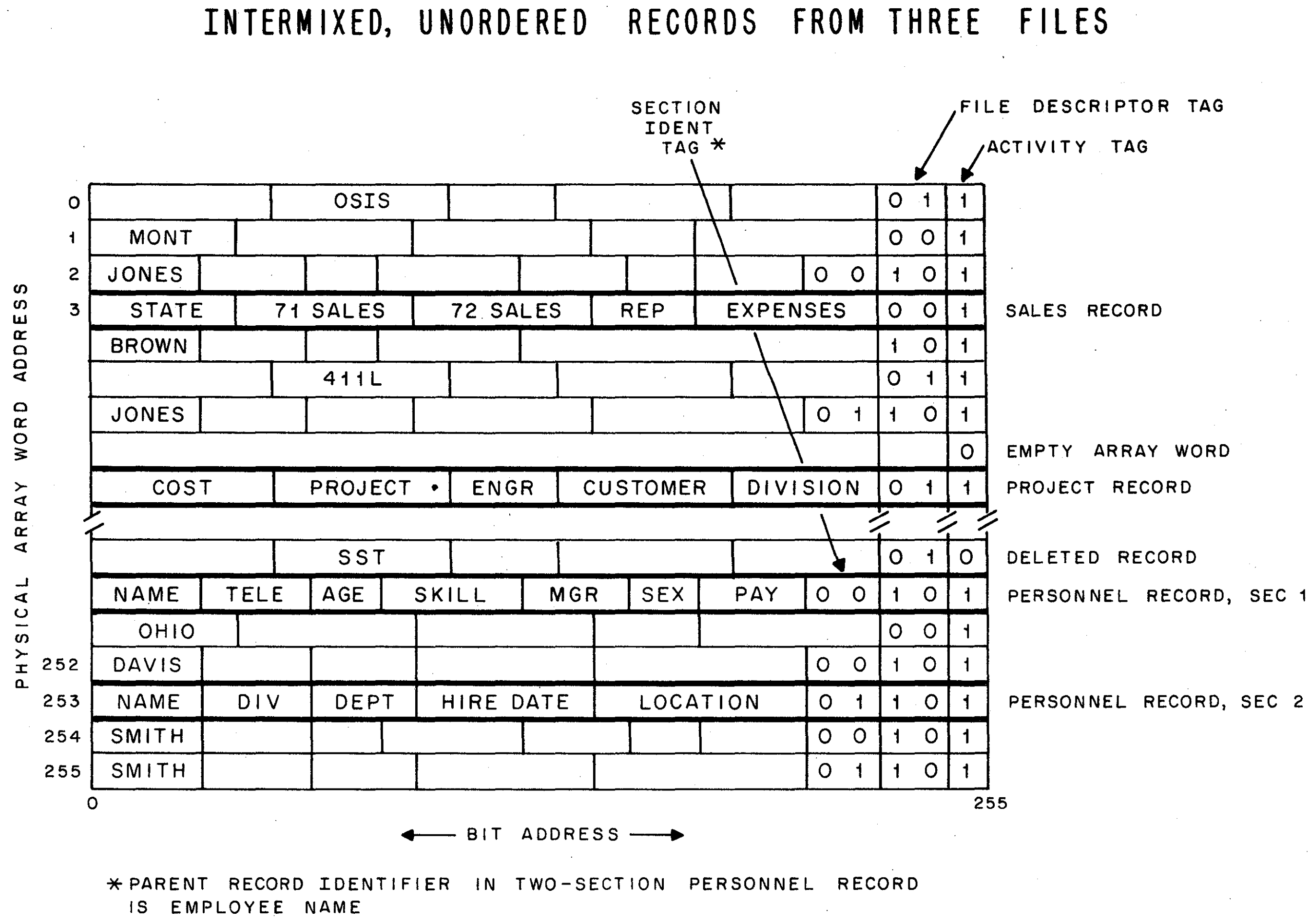


Figure 5—Associative array map example

the PE having the lowest physical address in the array (or arrays). The new record, with its activity field set to a "1," is written into this first empty location. The hardware pointer then moves to the next available empty memory location for writing another record if a batch of new entries must be loaded. If no empty locations are found the program will exit to whatever routine the programmer has chosen for handling this type of error—for example, if appropriate to a specific application, the program may select an age test of all records in a particular file, purging the oldest to make room for the newest. A record once located may be deleted from a file by merely setting the activity bit to an "O."

When a specific file is to be processed in some manner, the scattered locations containing the file's records are activated by performing EQC's on both the activity field and an n-bit "file descriptor" tag field. If, as in the example of Figure 5, the file descriptor field is two bits long, the entire selected file will be ready for processing in less than 2 microseconds (<1 μs for the activity bit search, <1 μs for the file descriptor field search).

Where record lengths are greater than the 256-bit length of the associative array word, several noncontiguous associative array words may be used to store the single record in sections, one section per array word. The format for each record section must contain the same activity and file descriptor fields as are used in all record formats, and in addition it must contain a parent record identifier and an n-bit "section identifier" tag field. The scattered locations containing the desired section of all records in the specific file may be activated by performing EQC's on the activity, file descriptor, and section identifier fields. All three searches can be completed in approximately 2 or 3 microseconds.

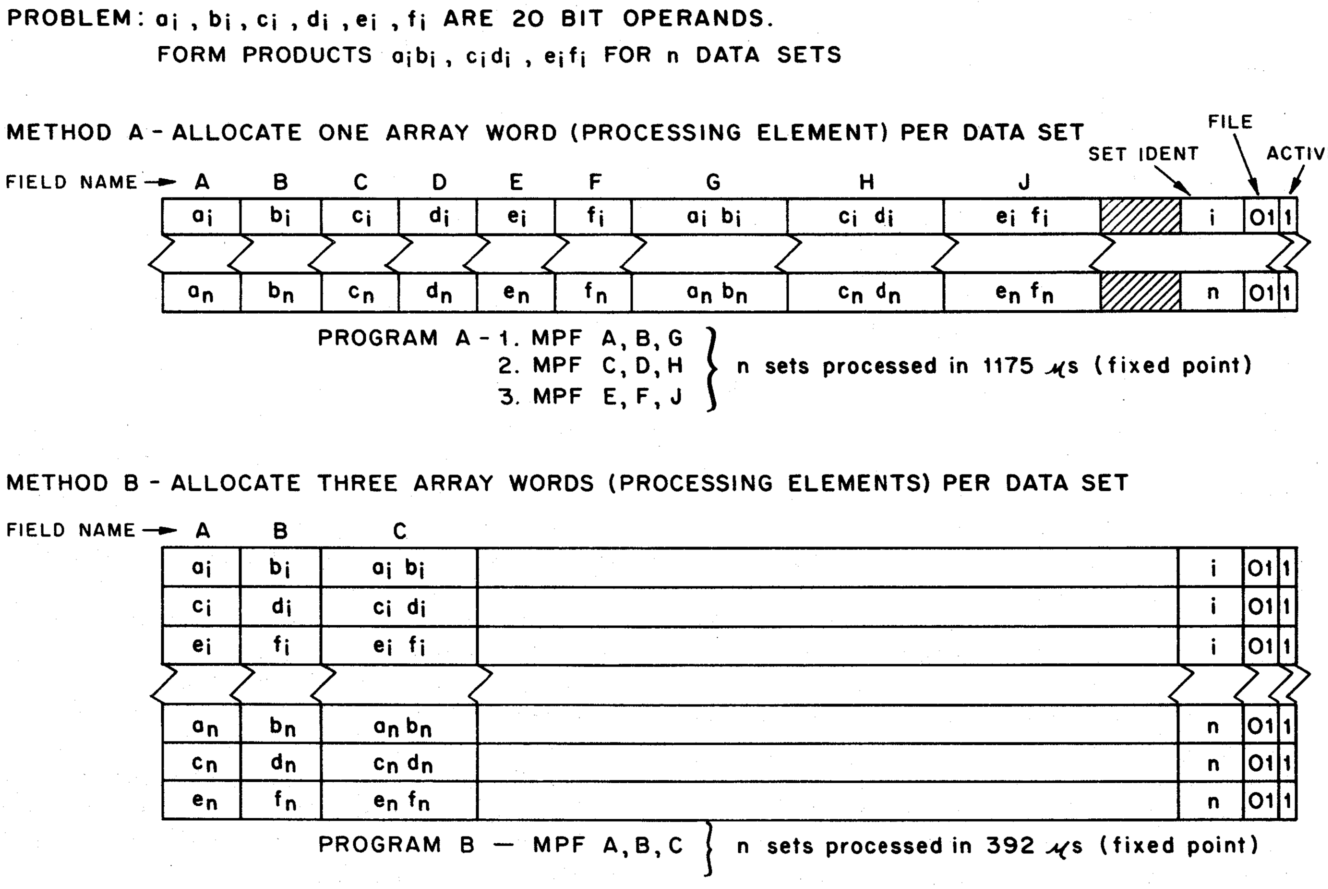
These two or three tag search operations in the AP permit random placement of records in the physical file and eliminate the bookeeping associated with file structuring and control required in conventional systems. The same approach is used for files which exceed the capacity of the associative arrays—the records of such files are stored in a similar manner on external mass storage devices and are paged into the arrays as required.

The strategy used to allocate array storage space can have a significant effect on program execution time. An example is shown in Figure 6 where the products of three operand pairs are required. In A, the operands are stored in a single array word. For 20-bit fixed point operands the three MPF instructions would execute in a total of 1175 microseconds. All similar data sets stored in other array words would be processed during the same instruction execution. However, an alternative storage scheme (B) which utilizes three PE's per data set requires only one MPF execution to produce the three products in 392 microseconds. If one thousand data sets were involved in each case the average multiply times per product would be 392 and 131 nanoseconds, respectively, but at the expense, in B, of using 3000 processing elements. Unused bits in B may be assigned to other functions˙

A last example of how array storage allocation can affect program execution time is shown in Figure 7 where the columns represent fields. Here the sum e_1, of 16 numbers is required. If the 16 numbers are directly or as a result of a previous computation stored in the same field of 16 physically contiguous array words, the near-neighbor relationships between the processing elements can be used to reduce the number of ADF executions to four. All similar 16 number sets would be processed at the same time.

STARAN APPLICATIONS

While many papers have appeared (see Minker[4] for a comprehensive bibliography) which discuss the application of AM's and AP's in information retrieval,



PROBLEM: a_i, b_i, c_i, d_i, e_i, f_i ARE 20 BIT OPERANDS.
FORM PRODUCTS a_ib_i, c_id_i, e_if_i FOR n DATA SETS

METHOD A - ALLOCATE ONE ARRAY WORD (PROCESSING ELEMENT) PER DATA SET

FIELD NAME → A	B	C	D	E	F	G	H	J		SET IDENT	FILE	ACTIVE
a_i	b_i	c_i	d_i	e_i	f_i	$a_i\ b_i$	$c_i\ d_i$	$e_i\ f_i$	▨	i	01	1
a_n	b_n	c_n	d_n	e_n	f_n	$a_n\ b_n$	$c_n\ d_n$	$e_n\ f_n$	▨	n	01	1

PROGRAM A - 1. MPF A, B, G
2. MPF C, D, H
3. MPF E, F, J
} n sets processed in 1175 μs (fixed point)

METHOD B - ALLOCATE THREE ARRAY WORDS (PROCESSING ELEMENTS) PER DATA SET

FIELD NAME → A	B	C				
a_i	b_i	$a_i\ b_i$		i	01	1
c_i	d_i	$c_i\ d_i$		i	01	1
e_i	f_i	$e_i\ f_i$		i	01	1
a_n	b_n	$a_n\ b_n$		n	01	1
c_n	d_n	$c_n\ d_n$		n	01	1
e_n	f_n	$e_n\ f_n$		n	01	1

PROGRAM B — MPF A, B, C } n sets processed in 392 μs (fixed point)

Figure 6—Effect of array memory allocation on execution time

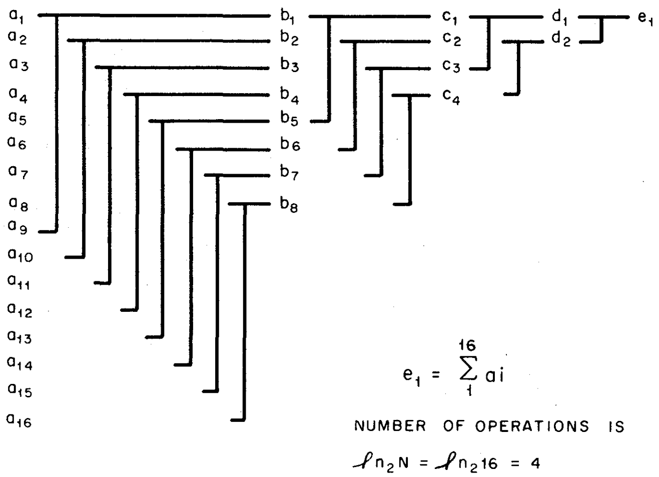


Figure 7—Tree-sum example

text editing, matrix computations, management information systems and sensor data processing systems, there are none yet published which describe actual results with operating AP equipment in any application. (But see Stillman: for a recent AM application result.)

Recent actual applications of the AP have been in real time sensor related surveillance and control systems. These initial applications share several common characteristics:

1. a highly active data base;
2. operations upon the data base involve multiple key searches in complex combinations of equal, greater, between-limits, etc., operations;
3. identical processing algorithms may be performed on sets of records which satisfy a complex search criterion;
4. one or more streams of input data must be processed in real time; and
5. there is a requirement for real time data output in accordance with individual selection criteria for multiple output devices.

A portion of the processing inherent in these applications is parallel-oriented and well suited to the array processing capability of the AP. On the other hand these same applications also involve a significant amount of sequentially-oriented computation which would be inefficient to perform upon any array processor, a simple example being coordinate conversion of serially occurring sensor reports.

Air traffic control

An example of an actual AP application in an air traffic control environment is shown in Figure 8. In this application a two array (512 processing elements) STARAN S-500 model was interfaced via leased telephone lines with the output of the FAA ARSR long range radar at Suitland, Maryland. Digitized radar and beacon reports for all air traffic within a 55 mile radius of Philadelphia were transmitted to STARAN in real time. An FAA air traffic controller's display of the type used in the new ARTS-III terminal ATC system and a Metrolab Digitalk-400 digital voice generator were interfaced with STARAN to provide real-time data output. The controller's keyboard was used to enter commands, call up various control programs and select display options.

Although a conventional computer is not shown explicitly in Figure 8 the sequentially oriented portions of the overall data processing load were programmed for and executed in the STARAN sequential controller as shown in Figure 9. Sequential and associative programs and instruction counts for STARAN are shown in Table II. In a larger system involving multiple sensors and displays, and more ATC functions such as metering and spacing, flight plan processing, and digital communications, the sequential and parallel workloads would increase to the point where a separate conventional computer system interfaced with the AP would be required.

The STARAN system was sized to process 400 tracks. Since the instantaneous airborne count in the 55 mile radius of Philadelphia was not expected to exceed 144 aircraft, a simulation program was developed to simultaneously generate 256 simulated

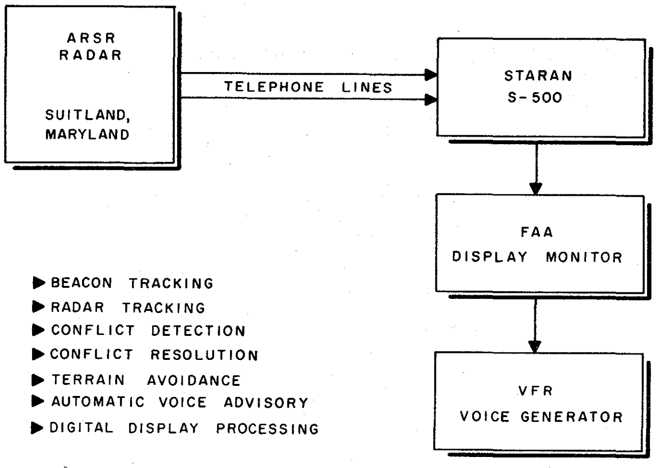


Figure 8—Air traffic control application

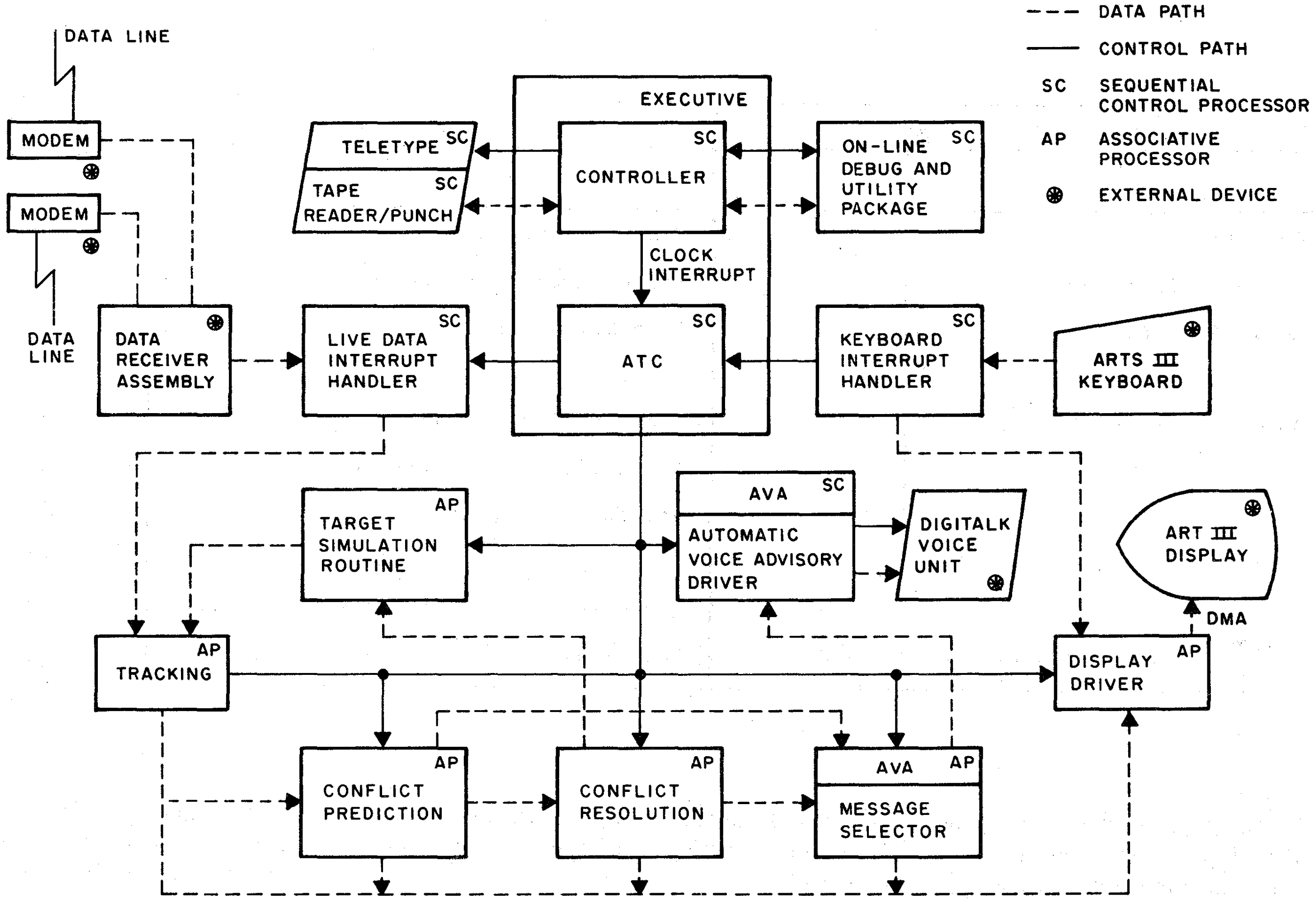


Figure 9—ATC program organization

aircraft tracks. Display options permitted display of mixed live and simulated aircraft. The 400 aircraft capacity is representative of the density expected as North-South traffic loads increase through the late '70s. Conflict prediction and resolution programs based upon computed track data were demonstrated and used to display conflict warning options. Automatic voice services were provided for operator-designated aircraft, thus simulating warning advisories for VFR pilots requesting the service. The voice messages, which in an operational system would be automatically radioed to the pilot, were generated by the Metrolab unit from digital formats produced by the associative processor and broadcast in the demonstration area via a public address system. A typical message would be read out in voice as, "ABLE BAKER CHARLIE, FAST TRAFFIC SEVEN O'CLOCK, 4 MILES, ALTITUDE 123 HUNDRED, NORTHEAST BOUND".

Top level flow charts for four of the associative programs used in the demonstration are shown in Figures 10, 11, 12, and 13. A detailed report is in preparation describing all of the ATC programs used in this demonstration, but some comments on the four flow charts shown may be of interest.

Live target tracking (Figure 10) is performed in two dimensions (mode C altitude data was not available) using both radar and beacon target reports to track all aircraft. Incoming reports are correlated against the entire track file using five correlation box sizes, three of which vary in size with range. Any incoming report which does not correlate with an existing track is used to automatically initiate a new tentative track. An aircraft track must correlate on two successive scans and have a velocity exceeding 21 knots to qualify as an established track and must correlate on three successive scans to achieve a track firmness level high enough to be displayed to a controller as a live target.

TABLE II—STARAN Air Traffic Control Programs

SEQUENTIAL PROGRAMS		ASSOCIATIVE PROGRAMS	
NAME	INSTR COUNT		INSTR COUNT
Executive	1600	Tracking System	881
Keyboard Interrupt		Track Simulation System	415
Real Time Interrupt		Turn Detection	88
Live Data Input		Conflict Prediction	488
Automatic Voice Output		Conflict Resolution	296
		Automatic Voice Advisory	709
		Display Processing	1140
		Total	4017
		Field Definition Statements Included	514
Net Operating Instructions	1600	Net Operating Instructions	3493

There are provisions for 15 levels of track firmness including 7 "coast" levels. If a report correlates with more than one track, special processing (second pass resolve) resolves the ambiguity. Correlated new reports in all tracks are used for position and velocity smoothing once per scan via an alpha-beta tracking filter where for each track one of nine sets of alpha-beta values is selected as a function of track history and the correlation box size required for the latest report correlation. If both beacon and radar reports correlate with a track, the radar report is used for position updating. Smoothed velocity and position values are used to predict the position of the aircraft for the next scan of the radar and for the look-ahead period involved in conflict prediction.

Track simulation processing (Figure 11) produces 256 tracks in three dimensions with up to four programmable legs for each track. Each leg can be of 0 to 5 minute duration and have a turn rate, acceleration, or altitude rate change. A leg change can be forced by the conflict resolution program to simulate pilot response to a ground controller's collision avoidance maneuver command. Targets may have velocities between 0-600 knots, altitudes between 100-52,000 feet, and altitude rates between 0-3000 feet per minute.

The conflict prediction program sequentially selects up to 100 operator-designated "controlled" or "AVA" aircraft, called reference tracks in Figure 12, and compares the future position of each during the look-ahead period with the future positions of all live and simulated aircraft and also to the static position of all terrain obstacles. Any detected conflicts cause conflict tags in the track word format to be set, making the tracks available for conflict display processing. A turn detection program not shown opens up the heading uncertainty for turning tracks.

Display processing (Figure 13) is a complex associative program which provides a variety of manage-by-exception display options and automatically moves operator-assigned alpha numeric identification display data blocks associated with displayed aircraft so as to prevent overlap of data blocks for aircraft in close proximity to one another on the display screen. Sector control, hand off, and quick-look processing is provided.

All programs listed in Table II were successfully demonstrated at three different locations in three successive weeks, using live radar data from the Suitland radar at each location. The associative programs were operated directly out of the bulk core and page 0 portions of control memory since there was no requirement, in view of the low 400 aircraft density

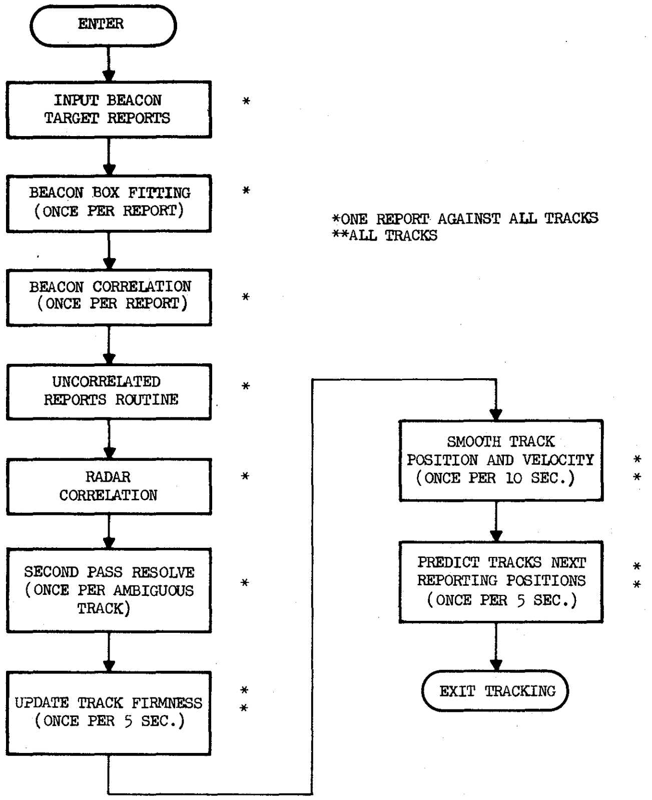

ENTER
INPUT BEACON TARGET REPORTS *
BEACON BOX FITTING (ONCE PER REPORT) *
BEACON CORRELATION (ONCE PER REPORT) *
UNCORRELATED REPORTS ROUTINE *
RADAR CORRELATION *
SECOND PASS RESOLVE (ONCE PER AMBIGUOUS TRACK) *
UPDATE TRACK FIRMNESS (ONCE PER 5 SEC.) **
SMOOTH TRACK POSITION AND VELOCITY (ONCE PER 10 SEC.) **
PREDICT TRACKS NEXT REPORTING POSITIONS (ONCE PER 5 SEC.) **
EXIT TRACKING
*ONE REPORT AGAINST ALL TRACKS
**ALL TRACKS

Figure 10—Live target tracking

involved, for the higher speed instruction accesses available from the page memories. At intervals during the demonstration all programs were demonstrated at a speed-up of 20 times real time with the exception of the live data and AVA programs which, being real-time, cannot be speeded up. Timing data for the individual program segments will be available in the final report. The entire program executed in less than 200 milliseconds per 2 second radar sector scan or in less than 10 percent of real time. All programming effort was completed in 4½ months with approximately 3 man-years of effort. This was the first and as of this writing the only actual demonstration of a production associative processor in a live signal environment known to the author. It was completed in June, 1972. Other actual applications currently in the programming process at Goodyear involve sonar, electronic warfare and large scale data management systems. These will be reported as results are achieved.

COST EFFECTIVENESS

Associative processor cost effectiveness can be expressed in elementary terms as shown in Figure 14 where performance is shown in terms of millions of instructions per second for the ADF and EQC instructions using two different operand lengths, and cost effectiveness is measured in terms of instructions per second per hardware dollar. This form of presentation was taken from Bell.[6]

Another cost effectiveness measure is to compare projected hardware and software costs of an associative configuration and an all-conventional design for the same new system requirements, where the associative configuration may include a conventional computer. Only a few attempts at this approach have been made to date and none have been confirmed through experience. One classified example, using a customer defined cost effectiveness formula, yielded a total system cost effectiveness ratio of 1.6 in favor of the associative configuration.

Of the two methods, the first is least useful because there is no way of estimating from these data how much of the associative computing capability can be used in an actual application. The second method is

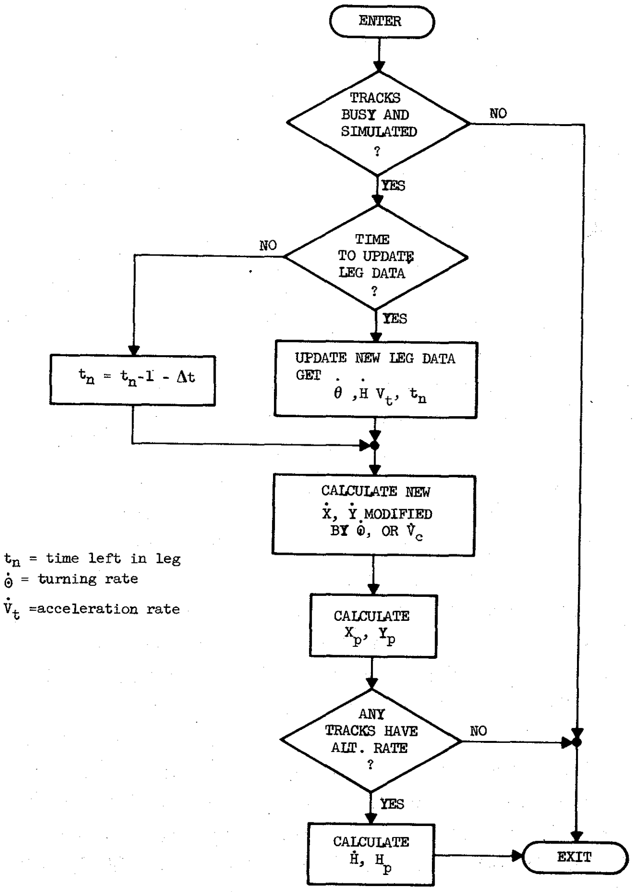

ENTER
TRACKS BUSY AND SIMULATED ?
NO
YES
TIME TO UPDATE LEG DATA ?
NO
YES
$t_n = t_n$-1 - Δt
UPDATE NEW LEG DATA GET $\dot{\theta}$, H $\dot{V}_t$, t_n
CALCULATE NEW $\dot{X}$, $\dot{Y}$ MODIFIED BY $\dot{\theta}$, OR $\dot{V}_c$
CALCULATE X_p, Y_p
ANY TRACKS HAVE ALT. RATE ?
NO
YES
CALCULATE $\dot{H}$, H_p
EXIT
t_n = time left in leg
$\dot{\theta}$ = turning rate
$\dot{V}_t$ =acceleration rate

Figure 11—Tracking simulation

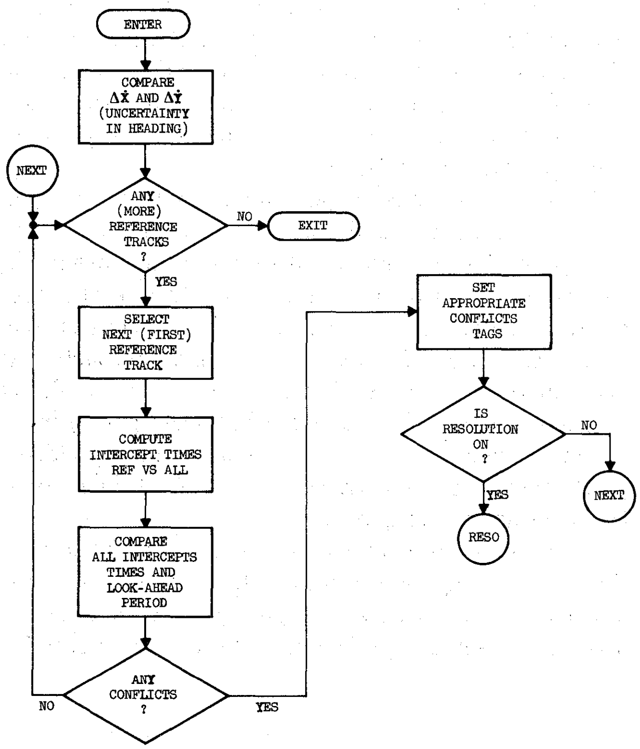

ENTER
COMPARE $\Delta\dot{X}$ AND $\Delta\dot{Y}$ (UNCERTAINTY IN HEADING)
NEXT
ANY (MORE) REFERENCE TRACKS ?
NO
EXIT
YES
SELECT NEXT (FIRST) REFERENCE TRACK
COMPUTE INTERCEPT TIMES REF VS ALL
COMPARE ALL INTERCEPTS TIMES AND LOOK-AHEAD PERIOD
ANY CONFLICTS ?
NO
YES
SET APPROPRIATE CONFLICTS TAGS
IS RESOLUTION ON ?
NO
YES
NEXT
RESO

Figure 12—Conflict prediction

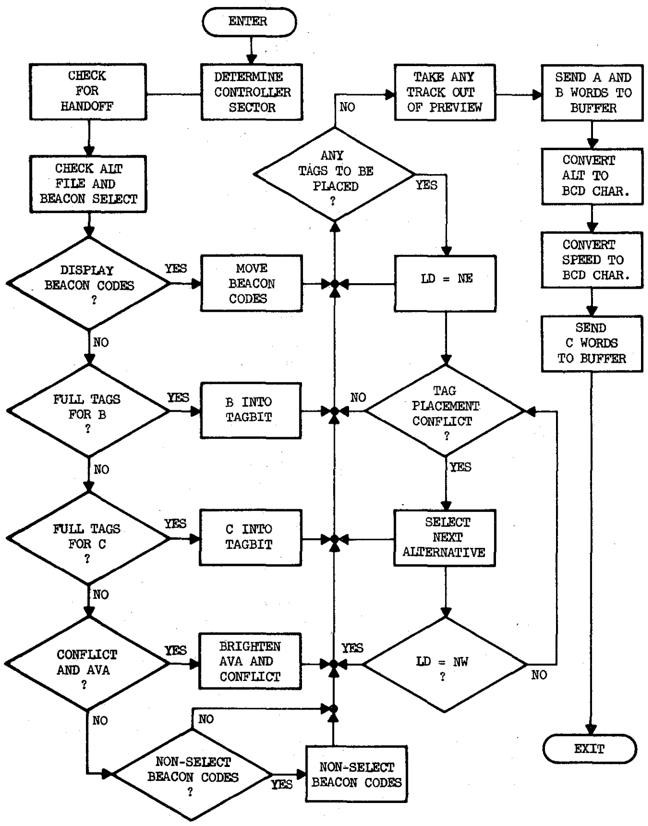
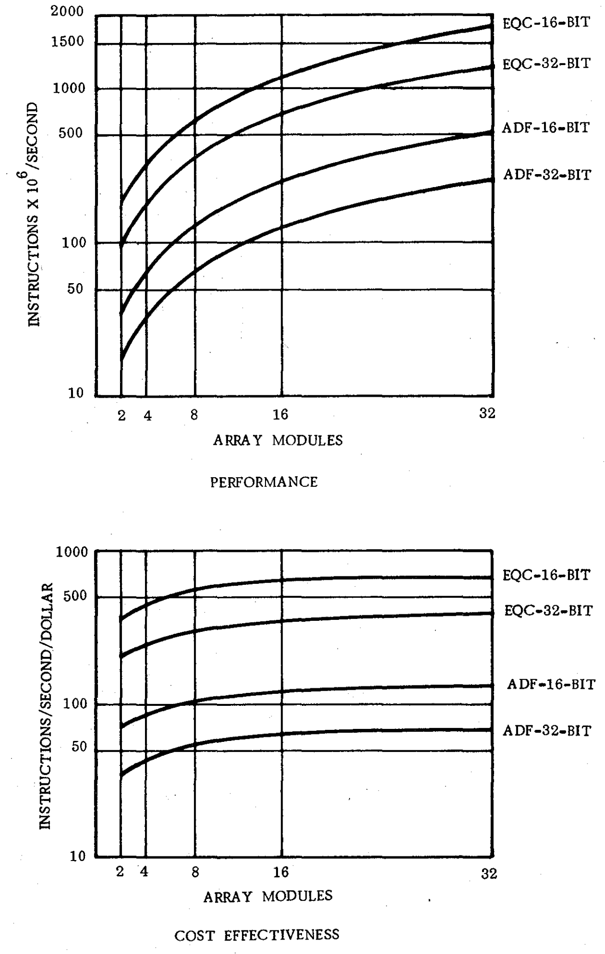


Figure 13—Display processing

more meaningful but is exceedingly expensive to use since it implies a significant engineering effort to derive processing algorithms, system flow charts, instruction counts, and timing estimates for both the conventional and the associative approach. The weakest element in this approach lies in the conventional approach software estimate which historically has been subject to overruns of major dimensions.



PERFORMANCE

COST EFFECTIVENESS

Figure 14—Array performance and cost effectiveness

A third method is to compare functional performance, hardware and software cost, growth capability and growth related costs, reliability, service and other pertinent aspects of two working examples of competing approaches to the same class of system application. Although it is a reliable method, it is not available at this time since no operational system of any kind has been implemented with an associative processor. The closest approach to it is the ATC demonstration described above but there is no similar conventional example to be found anywhere which includes the urgently needed large scale conflict detection processing included in the STARAN demonstration. On the other hand, an experienced ATC data processing system designer can appreciate the rapid solution time, small instruction count and low programming cost achieved with the STARAN for the troublesome high density tracking and display processing functions, but others not so well acquainted with ATC data processing problems may not find these data meaningful. This method also includes the benchmark test which is coming into regular use by the federal government in competitive large scale procurements of standard commercial equipment. Here again, however, due to the associative processor's recent arrival on the scene, no comparative performance data are yet available.

A fourth method, least useful in resolving the equipment selection and system design problems involved in a specific near term application, is based upon theoretical machine design considerations such as gate count ratios, logic to memory ratios and hardware efficiency or duty cycle ratios for conceptual machines which have not been reduced to practice during the typical seven year development cycle for new computer architecture.

Thus, until near term potentially cost effective associative processor applications are accomplished in operational environments, comparative cost effectiveness analyses of proposed associative versus conventional solutions will continue to be suspect. The next 12 to 18 months should produce a substantial improvement in the availability of reliable cost and performance data for associative processor applications.

SUMMARY

Although several manufacturers are developing associative processor equipment, the first version to be produced in a production configuration was introduced in May of 1972 by Goodyear Aerospace Corporation following FAA on-site tests in 1971 at Knoxville, Tennessee of a USAF-owned engineering model built and demonstrated by Goodyear in 1969.

The processor provides full content addressability and array arithmetic capability within "main frame" memory coupled with a unique capability for wide bandwidth (over 3000 megabits/sec for a 4-array STARAN) input-output data transfers to mass data stores. The associative programming language, APPLE, provides a flexible and convenient assembler for programming array arithmetic and search algorithms without the complex and costly indexing, nested loop and data manipulation constructions required in conventional computer programming.

The associative processor may be viewed as a software-programmable super-peripheral, or special purpose subsidiary processor, for attachment to any general purpose conventional computer system via standard channel attachment. In this role the super-peripheral is assigned parallel oriented problem segments and data bases which would otherwise, through excess operating system software overhead, tend to choke the conventional machine.

Although first applications of the associative processor are of the real time, dedicated, command and control type, the extension to large scale data base management, on-line management information systems with immediate response to complex multiple-key queries, and large scale matrix computations await only user decision and ingenuity to accomplish now that production hardware and software has become available at the 370/145 price level.

The cost effectiveness of associative processing has yet to be proven in operational systems, but test results from initial users should accumulate rapidly now that associative processing is no longer only an interesting concept in the literature.

REFERENCES

1 A E SLADE H O McMAHON
The cryotron catalog memory system
Proc 1957 FJCC Vol 10 pp 115-120

2 L C FULMER W C MEILANDER
A modular plated wire associative processor
Proc IEEE Computer Group Conference June 1970

3 J A RUDOLPH L C FULMER
W C MEILANDER
The coming of age of the associative processor
Electronics February 15 1971 pp 91-96

4 J MINKER
A bibliography of associative or content-addressable memory system: 1956-1971
Auerbach Corporation 121 N Broad Street Philadelphia Pa 19107 June 15 1971

5 N J STILLMAN
Associative processing and computer graphics—A feasibility study
USAF Report RADC-TR-72-57 April 1972

6 C G BELL R CHEN S REGE
Effect of technology on near term computer structures
Computer March-April 1972 pp 29-38

SIFT—Software Implemented Fault Tolerance

by JOHN H. WENSLEY

Stanford Research Institute
Menlo Park, California

INTRODUCTION

Many computer applications have stringent requirements for continued correct operation of the computer in the presence of internal faults. The subject of design of such highly reliable computers has been extensively studied,[11-14] and numerous techniques have been developed to achieve this high reliability. Such computers are termed "fault tolerant"; examples of applications are found in the aerospace industry, communication systems, and computer networks. Several designs of such systems have been proposed[2,5,8,11,12,13] and some have been implemented. In general, these designs contain extensive hard-wired logic for such functions as fault masking, comparison, switching, and encoding-decoding.

This paper describes a new approach to the design of a fault-tolerant computer, with strong emphasis on software techniques to achieve fault tolerance and corresponding deemphasis on special hardware units. One characteristic of the particular software approach taken is that erroneous results are not detected immediately after they occur, but rather at the conclusion of the processing of a task. However, the errors are not permitted to propagate.

The particular design discussed here is tailored to the use of computers for control functions in an advanced technology transport aircraft; this application determines the scale of the proposed system. Although extension to other applications might change the size or speed of the system (or its units), the basic concepts have sufficient generality to cover many applications.

In designing the system, a basic consideration is that the advent of large-scale-integrated (LSI) circuits implies that any reconfiguration or discarding of equipment should be carried out at the unit level (CPUs or memory blocks) rather than at the component level (gates or registers). In addition, economic use of LSI demands that the number of different types of units be minimized, with high replication of each type.

Fault-tolerant computer systems vary greatly in reliability requirements. A typical requirement in space applications is for a probability between 95 percent and 99 percent that computing capability will exist after 5 to 10 years of operation. This implies a mean time between failure (MTBF) from 100 to 1000 years.* In the control of an aircraft, to which this design was aimed, the requirement was for a probability of failure less than 10^{-8} during a 10-hour operational period. This translates to a MTBF of 10^4 years, i.e., 10 to 100 times more stringent than the above. The consequence of failure (possible loss of human lives and economic loss) is, in this application, extremely high and justifies the use of extensive redundancy in the computer system where cost is, even with redundancy, a small proportion of total aircraft cost. The computing load for this application is such that the computer must have approximately 16K words of memory and be capable of better than 0.5 MIPS.** Assuming LSI circuitry with a chip failure probability of 10^{-6} per hour, the overall system design must assume correct functional behavior in the presence of multiple chip failures, which can be expected in a computer system containing several hundred LSI chips.

We are concerned with faults in the two major subsystems, i.e., the processor and the memory. With reasonable predictions concerning LSI development in the next few years, analysis shows that the processor will require approximately 10 percent of the chips required for the memory. Therefore, we regard

* This statement does not imply that a single computer will survive for 100 to 1000 years, but that n such computers will, after y years, have suffered $n \cdot y/100$ or $n \cdot y/1000$ failures.

** Millions of instructions per second.

replication of the processor as an economic checking and fault-masking technique. Protection of the memory function can be carried out either by replication or coding or by a combination of both. This paper describes a system using memory replication, but the basic concepts are compatible with alternative methods for protecting the memory.

The important features of the system can be implemented by a range of techniques going from hardware, through microprogram, through system software, to application software. The computer system described in this paper places heavy emphasis on the use of software to carry out fault detection and correction procedures. The fault-tolerant procedures can be made transparent to the application programmer by suitable design of the support software such as compilers or assemblers. A system in which such functions are achieved by suitably designed hardware (or microprogram) is also possible.

A central feature of the described system is the prevention of fault propagation by the use of read-only connections between processing modules. Another important feature is the avoidance of any need for a "lock-step" operation of replicated units, and a reduction in the frequency of fault checking. This results from the strategy of only checking when the state of the controlled (aircraft) system changes rather than at each change of state of the computer. An implementation in which more of the fault-tolerant features were in hardware would be faster in operation at the expense of flexibility of change that is given by the software implementation as described. The system as presented gives the designer the freedom to tailor the system to the application by the following important trade-off possibilities:

- An increase of speed by placing more of the fault-tolerant functions in hardware or microprograms.
- Flexibility for varying the amount of protection given to different application programs by using software fault-tolerant techniques.
- Ability to change the fault-tolerant strategies as new technologies emerge with new reliability characteristics.

BACKGROUND

Existing designs of fault-tolerant computing systems use a variety of redundancy techniques to achieve fault tolerance. These techniques include, for example, special codes for error detection and correction, and the replication of units with means for detecting whether or not several units carrying out the same operation are in agreement.

The JPL STAR computer[2] uses several redundant codes at different parts of the system, as well as special-purpose hardware to perform checking and rollback. The approach taken by Hopkins[11] does not include the extensive use of redundant codes but relies heavily on replicated CPUs, busses, and memories, with special units to check for agreement between units. These and other systems are designed to detect errors soon after they have occurred—usually before the state of the CPUs has been irreversibly changed or a memory cell has been overwritten. These systems require that any replicated units must stay in close step with each other, usually at the instruction level (so-called lock step). When detected, a faulty unit in these systems is removed from the system (e.g., by switching or removing power). If no provision is made to return a once faulty unit to service when it returns to correct operation, a severe cost penalty is incurred in the event of transient errors.

The system we describe has many properties that, in total, distinguish it from other fault-tolerant systems.

- Replicated units do not operate in lock-step mode, but are only loosely synchronized. The communication between CPUs is asynchronous, thereby removing the need for an ultrareliable system clock.
- Agreement between replicated units is verified only at the completion of program segments (tasks).
- Faulty units are not necessarily removed but can either be ignored or assigned to tasks having no overall effect.
- Transient faults do not necessarily cause permanent removal of the faulty units. Furthermore, the looseness of synchronization among sets of tasks makes it possible to enhance immunity from transients, by providing that redundant versions of a computation may be done at different moments in time.
- The degree of fault tolerance can be different for different tasks being performed, and can be different at different times for the same task.
- No special hardware is used to carry out fault detection or correction.
- Communication between CPUs is minimized so that low bandwidth busses can be used, thereby facilitating physical separation of modules in environments where physical damage is a hazard.
- The design concept is independent of the way in

which the units are built, i.e., no specialization of CPU or memory design is required for fault tolerance, thereby allowing the choice to be based on other properties, e.g., speed, availability.
- The total computing power of the system can be varied by using units of different speed or by changing the number of units.

SYSTEM DESIGN

The system (Figure 1) consists of a number of modules, each composed of a memory and a processing unit. The individual processing units within the modules are connected to the corresponding memory units with wide bandwidth busses. The intermodule bus organization (B_1,B_2,B_3)* is designed to allow a processor to read from any memory but not to write into other memory units. The intermodule bus is expected to have a much lower bandwidth than an intramodule bus.

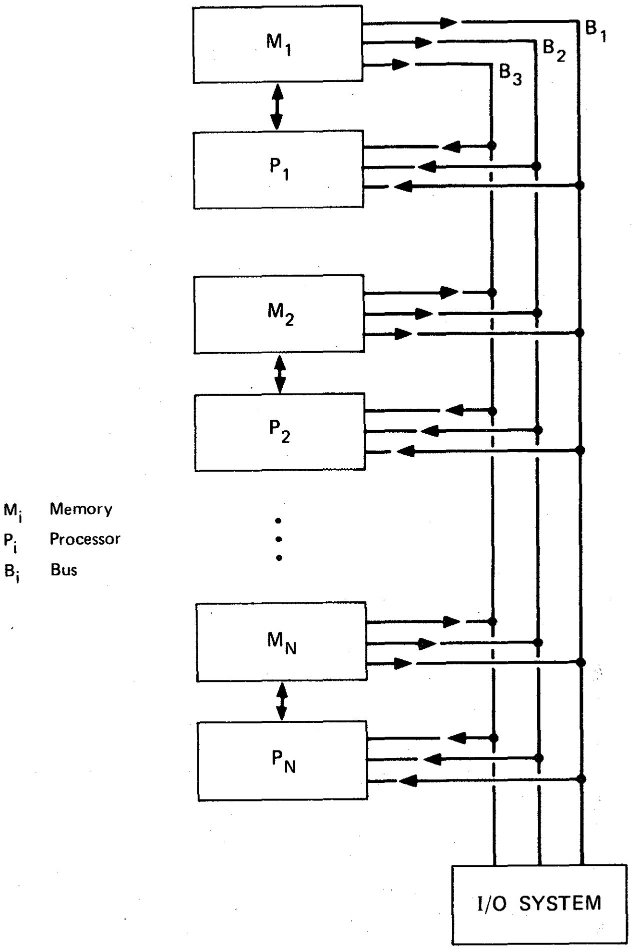
The input/output (I/O) system, discussed in a later section, is assumed to be connected to the busses B_1, B_2, and B_3. The input/output system shown in Figure 1 consists of all the noncomputing units, e.g., transducers, actuators, and sensors. The part of the total input/output that is carried out by program, e.g., formatting or code conversion, is handled in the same manner as any other task, i.e., is replicated in several processors.

The system is viewed as being regular in that no module is *a priori* assigned a special role. All computations that require high reliability are carried out in several modules. We assume for the purpose of this description that critical tasks are processed in three units.

The computations that must be carried out are broken into a number of tasks in such a way that no task requires more computing power than can be supplied by one processor. The tasks are given the designations, A, B, C . . .; the processors are numbered 1, 2, 3. . . . Each processor is capable of being multiprogrammed over a number of tasks, as illustrated in Figure 2.

The control of the computing system is carried out by an executive system that can be segmented by function into two parts:

(1) Local Executive: functions that apply to each processor (e.g., dispatching,* reporting errors, loading new task programs).
(2) System Executive: functions that are global to the system (e.g., allocation and scheduling of work load, reconfiguring).



TA-710522-220

Figure 1—System configuration

A complete set of the software functions of Class 1 is present in each processor; those in Class 2 are carried out in a sufficient number of processors to provide the degree of fault tolerance required. The functions are realized by programs that have the same task structure as all other programs.

The normal operating mode for a processor carrying out a task is to follow the flow of control shown in Figure 3. Data required for the task are assumed to have been computed by several processors (including

* The bus logic envisioned does not use voting. The number three is chosen for convenience of discussion.

* Dispatching is the executive function that initiates a new task at the completion of the previous one.

TASKS \ PROCESSORS	1	2	3	4	5	6	• • • n
A	X	X		X			
B			X		X	X	
C		X					
D	X		X	X			
E		X		X	X		
F			X		X	X	
G	X	X	X				
H	X			X		X	
I	X		X		X		
J	X	X	X	X	X	X	
• • • N	• • •						

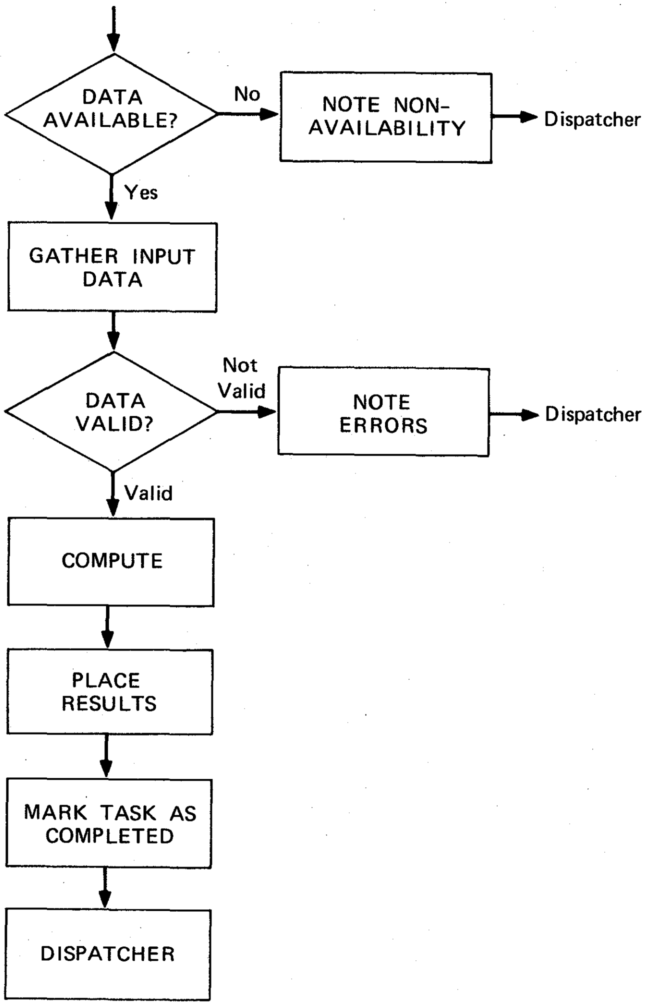
TA-710522-221

Figure 2—An example of task/processor allocation

possibly the same one carrying out the task). A check is made to see if the data are available in all processors. If not, the fact is noted in the memory of the module and the dispatcher program within the module is entered to determine the next task to be processed. The next processing is the reading of input data from the several processors where copies exist. A validation is now carried out, typically by a two-out-of-three vote. If any of the copies of the input data are found not to agree, then this fact is noted for later processing by the executive. If all the copies are different, the fact is noted and control moves to the dispatcher program. The computation of the task is now carried out, the results are left in the memory of the module, and note is made (in the module) of the fact that the task is computed.

Certain important principles apply in the above scheme:

- No processor writes into the memory of another module.
- Input data in a module are not destroyed during the computation: If the computation is repetitive, the results of one cycle that may be used as input for the next cycle are placed in a different location in memory. Similarly, because the input data within one module may be needed later by another processor carrying out the same task, the input data must not be destroyed until all cooperating processors have read, validated, and used the data. This may require that the data be preserved over several iterations if they are used by another task that is delayed behind the first.
- All conditions (e.g., errors, task complete) are left as notes to be read later by the system executive.
- The dispatcher program, which exists in each module, maintains a queue of tasks to be computed. The data for this queue are read from the memories of the modules that are running the executive. The flow of control of the dispatcher is



TA-710522-222

Figure 3—Typical task flow

itself similar to that shown in Figure 3, except at the end, when the control is transferred to the task that is at the head of the queue.

- The dispatcher in each processor checks from time to time to see if the system executive has changed the queue of tasks for that processor. A single bit (per processor) is set in the system executive tables to indicate a change of the queue. If this bit is not set, the dispatcher waits some time (e.g., 1 msec.) before querying it again, thereby preventing continuous interrogation and consequent heavy intermodule bus traffic.

The above scheme has a degree of fault tolerance without special hardware requirements on the memory or processor units. In particular, an erroneous calculation carried out by a module does not destroy the validity of the total system, because results are rejected by the next calculation.

The general strategy outlined above places certain constraints on hardware and software components. These constraints are discussed below.

INPUT/OUTPUT

The input/output subsystem must be designed and operated with the same fault tolerance as the central processing complex. Different modes of operation are possible, depending on the devices that are connected to and controlled by the system. The favored principle is to use replication wherever possible. Varying capabilities of fault tolerance in the central computing system can be achieved by using varying replication and by voting at all times when valid data are required (e.g., at the start of a task). The results of a calculation will exist in several (usually three) copies and eventually a vote must be taken. A vote that is required to allow another task calculation is carried out in multiple modules; however, if the vote is for output, then the output system or output unit must conduct the final vote.

There are circumstances where the nature of the input/output unit assists fault tolerance through replication, as in the following cases:

- Certain input systems (sensors) can be replicated; each sensor is then individually read (and voted on) by all modules requiring the input.
- Certain output devices can be built in a way that employs a "natural" kind of voting process in the final output medium. For example, a CRT display could be refreshed with each frame derived from a different module. Data on which all modules agreed would be displayed brightly; other data would be more faint. Assuming that faults persist only for short periods, this would result in a temporary flicker for a few frames before the executive removed the malfunctioning module from the calculation. In the application to which the design is aimed, there are other output devices, e.g., flap controls possessing similar "natural" voting capabilities.

In the event that the device is not in one of the above classes, another "final voter" must be designed that inherently possesses the required reliability. This consideration is independent of the architecture chosen for the central computing system.

We note that the architecture described here can operate in a mode whereby the replicated versions of output data (or the replicated data from input sensors) can be processed by any of the processing modules; hence, no modules need be specially designed for this function.

BUS DESIGN

The bus system (B_1, B_2, B_3 as in Figure 1) used for communication between modules must be designed to be fault tolerant. We remind readers that the bus system is used only to allow the processor of one module to read from the memory of a different module. The design need not be such that all bus traffic is checked (as in most other fault-tolerant architectures); however, it should allow a processor the choice of different busses in the event that a bus has failed.

A structure based on a four-port memory module is shown in Figure 1. In this structure, each module would have connection between its units (processor and memory). The bus structure, B_1, B_2, B_3, would enable a processor to choose different paths in reading data from the memory units of different modules. It would be appropriate to connect the I/O system to this bus structure. In the event that a four-port memory unit such as shown in Figure 1 is not available (or not suitable from other standpoints), then the structure can be achieved by attaching a single-port memory to all busses using conventional techniques.

A processor that needs to read from the memory of a different module must seize control of a bus. Logic associated with a bus must ensure that only one processor has control of a bus at any time. In addition, the bus must be allocated to a processor for only a finite time, thereby preventing a faulty processor from seizing a bus permanently. An internal clock in each bus can control the period for which the processor in question dominates a bus. A failure in this control logic only causes the loss of that bus. It remains to be shown that no situations can occur where the failure of one unit can cause incorrect action of several other units, i.e., we require a design so that faults remain localized.

The interconnection of the units has only one purpose—to enable any processor to read data from any memory using any bus. The interconnection system does not allow a processor to write into other memories. A separate connection is assumed for a processor to write into its own memory.

In summary, the following sequence of action is carried out in reading data word (w) from memory (m) to processor (p) via bus (b).

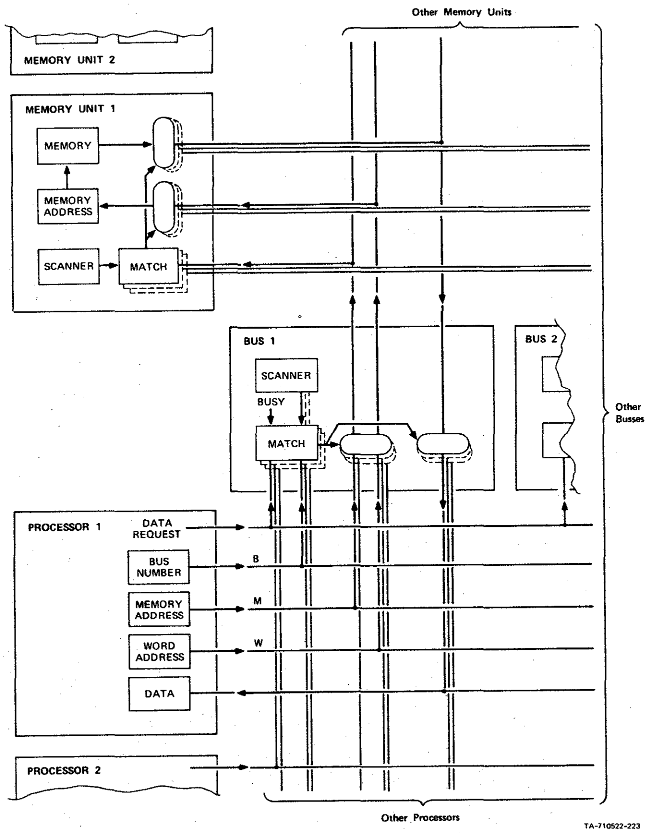
(1) Processor p places b, m, and w in registers and signals all busses with a DATA REQUEST.
(2) All nonbusy busses scan all processor DATA REQUEST lines (continuously).
(3) If a data request line is on, and b equals the bus number, the bus goes into BUSY state and stops scanning the processors. The requested bus has now been selected by the processor.
(4) The selected bus transmits m, w, and DATA REQUEST from the processor registers to all memory modules.
(5) All nonbusy memory modules scan all busses for a DATA REQUEST line that is on, and then compares the m on that bus with its own number.
(6) If a match is found, the memory goes into BUSY state and ceases scanning the busses. The w on the bus is placed in the memory address register and a read request issued to the memory. The memory is now selected.
(7) When the word is read by the memory, it is placed on the data lines of the bus and a DATA READY line is turned on.
(8) The DATA READY and data are transmitted to the requesting processor. When the data have been received by the processor, the DATA REQUEST line from that processor is turned off.

(9) Action 8 above will cause the BUSY states (actions 3 and 6) to be dropped, and the bus and memory to resume scanning for other requests.

In the above sequence, each unit that requests action of another unit makes a request (e.g., DATA REQUEST). The granting of the request is made by the requestee. This arrangement, for example, will prevent a processor from requesting all the busses simultaneously as the busses will only respond if the bus request (b) agrees with their bus number. Therefore, it would require failure of all of the busses to completely disable the bus structure.

In addition to the above, it is assumed that each unit has logic associated with it that prevents it from being seized indefinitely. This logic, in effect, says "If I have been BUSY for greater than a time interval Δ, then the particular connection will be broken and scanning will be resumed for other units requiring action." It is possible to incorporate in this logic the capability to ignore requests from the offending requestor in the future, thereby removing that unit from affecting further system operation. The time interval, Δ, will be chosen to be just greater than the greatest time of any correct action request.

The word address (w) that is transmitted to the memory module can be subject to any transformation that is convenient in the design of the processor or



TA-710522-223

Figure 4—Processor/bus/memory connection

memory, i.e., it can include indirect addressing, indexing, base registers, paging, or any convenient combination of these. In addition, it is possible to incorporate a cache (in the IBM 360/85 sense) in the processor design.

The scheme outlined above obviates the need to provide a BURST MODE type of transmission as each word that is transferred can follow the sequence given. In the event that several words are required, the processor successively requests each word and the bus is seized and the word is delivered. If other processors require the use of the bus during the period of the multiple word transfer, a form of cycle stealing will take place as the bus scans the other units and honors the request before resuming scanning.

A suitable structure for the processor/bus/memory connection is shown in Figure 4.

THE SYSTEM EXECUTIVE

The system executive is concerned only with allocation of resources. All other special functions typically associated with an operating system (e.g., I/O control) would be treated as parts of the application program set.

It is expected that, in steady state, the executive would employ a simple scheduling algorithm to allocate resources. Exceptions to this would occur under two conditions:

(1) Change of task set to be computed
(2) Error conditions—either transient or permanent.

In both of the above cases it becomes necessary to reallocate resources. This task would not have to be carried out with high speed because, in the application considered, condition 1 above will be known in advance; condition 2 can be delayed for a short time because of the fault-tolerant procedure of replication and voting, which is carried out by the processors.

The executive system carries out any required synchronization. For example, the calculation required for advanced attitude and flutter control in certain aircraft must be carried out every four milliseconds. This calculation entails reading some sensors and then computing the new state variables using the old state variables and the input data. All modules assigned to this task have queues in their dispatcher task. The executive places a message in its memory for these processors to update their queues, whereupon the next calculation of this task is carried out by the several processors. When tasks are assigned to processors, the executive must designate the other cooperating processors so that all data required may be obtained. For the executive to carry out this synchronization, it must have a time reference that can be read by the processors or that causes an interrupt.

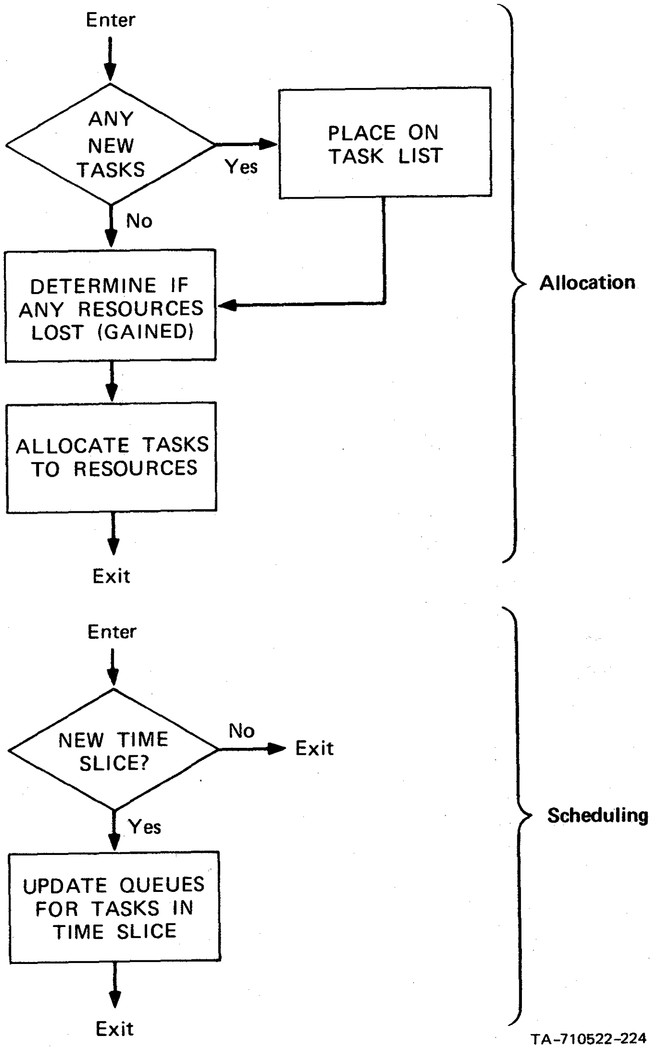


Figure 5—Executive

The calculations carried out by the executive are handled in exactly the same manner as other calculations (see Figure 3). A number of processors cooperate on this task, thereby providing fault tolerance when computing the executive. All processors within the system must know the designations of the several processors that are assigned to the executive. These data are held in the memory associated with the dispatcher that requires input data from the executive.

The flow of control for that part of the executive concerned with allocation and scheduling is shown in Figure 5. This flow will be embedded as the actual calculation as shown in the fourth ("Compute") box of Figure 3. The allocation function is used to determine which tasks are to be computed and by which modules. It will be invoked relatively infrequently as it is required only when allocation changes are to be made. The scheduling function determines the time period during which any calculation is carried out.

FAULT-TOLERANCE PROCEDURES

By suitable design of the executive, the system architecture can carry out different fault-tolerance procedures for different requirements. The assumed fault-detection method is by comparison of multiple copies of data. This comparison is carried out by software imbedded in a system routine—a copy of which is present in all processors.

The first step in the computation of any task is to input the data required to carry out the task. This data will exist in the memory of three computing modules. We will use the phrase "Input Data Set" (IDS) to denote the set of words required to carry out the calculation of a task. It is envisioned that all tasks requiring data will obtain it by calling a single subroutine. This subroutine is the only code (outside the executive) that is concerned with detecting errors; correcting them, in some cases; and in all cases, reporting errors to the executive. By the use of a single subroutine for error detection, avoidance, and reporting, the application programmer is relieved of the concern for this aspect of the system. This subroutine is shown in flow chart form in Figure 6. Its functional specification is explained in terms of the input parameters, output parameters, and the actions carried out.

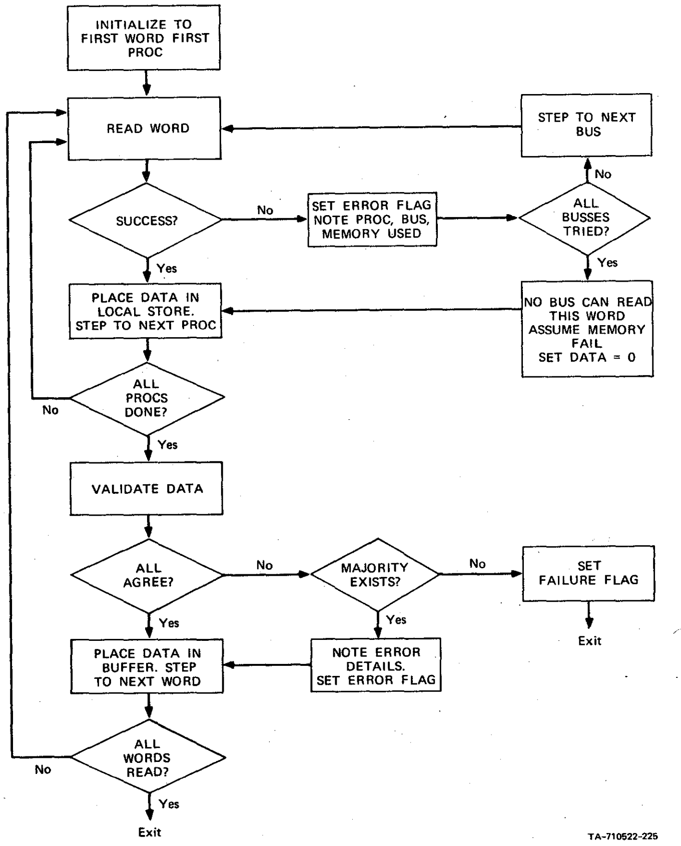

INITIALIZE TO FIRST WORD FIRST PROC
READ WORD
STEP TO NEXT BUS
SUCCESS?
No
SET ERROR FLAG NOTE PROC, BUS, MEMORY USED
ALL BUSSES TRIED?
No
Yes
Yes
PLACE DATA IN LOCAL STORE. STEP TO NEXT PROC
NO BUS CAN READ THIS WORD ASSUME MEMORY FAIL SET DATA = 0
ALL PROCS DONE?
No
Yes
VALIDATE DATA
ALL AGREE?
No
MAJORITY EXISTS?
No
SET FAILURE FLAG
Yes
Yes
Exit
PLACE DATA IN BUFFER. STEP TO NEXT WORD
NOTE ERROR DETAILS. SET ERROR FLAG
ALL WORDS READ?
No
Yes
Exit
TA-710522-225

Figure 6—Input communication subroutine

Input parameters

IDS NUMBER (The identification of which input data set is to be input.)

IDS SIZE (The number of words to be input.)

BUFFER (The address of the buffer in which the words are to be placed.)

PROC LIST (The address of a list of processor numbers from which to input.)

Output parameters

FAILURE FLAG (A boolean output variable, set = 1 if the input could not be accomplished.)

ERROR FLAG (A boolean output variable, set = 1 if input was successful but an error was detected.)

ERROR VECTOR (The specification of the IDS, word position, bus and memory involved in an erroneous input.)

Action

Read an input data set (IDS NUMBER) consisting of IDS SIZE words from the processor memories specified by PROC LIST. If all versions of each word obtained from the different processor memories agree, the data are placed in the memory at address BUFFER, the ERROR and FAILURE FLAG are set to 0, and a return is made to the calling program. If all versions of a word do not agree but a majority agreement exists, the data are placed in the BUFFER, the ERROR FLAG is set to 1, and the details of the (presumed) erroneous input are placed in memory to be read later by the executive. If no agreement can be found, the ERROR and FAILURE FLAGS are set to 1, the data are not placed in the BUFFER, and a return is made. If no action can be accomplished (e.g., because of a faulty bus system) the units that are faulty are noted, and a return is made. The subroutine will attempt to use different busses for each word transferred. If no response is obtained from an input request, the subroutine steps to the next higher bus.

Fault detection by software voting is compatible with hardware fault-detection techniques such as parity schemes. Such hardware, if it exists in memories, busses or processors, can be used to assist detection and diagnosis of faults. The primary advantage of incorporation of hardware checking is to allow faster checking in the event that an application requires faster correction of fault conditions than can be achieved by software.

An important benefit in using software techniques for fault detection and tolerance is that freedom is retained to change the degree of fault tolerance, either because experience gives data on which better methods can be based, or because the different applications require different degrees of fault tolerance, i.e., some are more critical than others.

If threefold replication is used throughout the system, a single faulty unit will result in one of the replicated processes computing a wrong result. The use of the wrong result in subsequent calculations will be avoided by the fact that other (correct) copies of the data will exist in other modules and when used will, by voting, enable a processor to distinguish the correct data from that which is erroneous.

Consider now the case of double faults existing simultaneously. We must distinguish two cases, uncorrelated and correlated faults. By correlated faults, we mean two faults that cause the computation of two equal but incorrect results. Clearly, two correlated faults cannot be tolerated if the fault-tolerance procedure consists merely of voting among three versions of all results. The probability of such correlated faults will be extremely low and for most applications is acceptable. However, in the system as described, we can achieve greater reliability in the event that the application is so critical that this low probability is still unacceptable. Two strategies are:

(1) Use threefold replication for all critical applications, and in the event of *any* disagreement, do not use the results until yet further processors have carried out a repetition of the calculation, for example use two more processors (making a total of five) and only act if three or more agree.

(2) Use fivefold (or greater) replication* of tasks for all critical applications.

Both of the above strategies will prevent double correlated errors from causing the use of a wrong result in subsequent programs or output. The cost penalty in the above strategies implies that they will only be used for extremely critical applications, where the cost of extra computing equipment is small compared with the penalty for failure, e.g., in aircraft and space missions.

In the case of double uncorrelated faults, we need only consider the case of simultaneous faults. Double faults that occur separated by a time sufficient for the executive to have carried out corrective action after the first fault do not need to be regarded as different than two instances of single faults, which can be tolerated.

Two simultaneous but uncorrelated faults will have the possible effect of producing two different incorrect results from a calculation. These two results will be compared with the one correct result produced by the nonfaulty unit in a threefold replication scheme. Before the result is used in any subsequent calculation (or output), the presence of three differing results will be detected and the executive will initiate greater replication in other processors until sufficient agreement can be found to distinguish the correct from the incorrect result.

In considering the effect of multiple faults in the system, an improvement in reliability is achieved by the fact that the multiple processors are not operating in a lock-step mode. A short term, widespread transient in the system hardware (e.g., power supply or bus system) will not necessarily cause errors in the same application programs in the processors, thereby increasing the probability of being able to detect and correct the errors from the transient.

The executive of the system must itself be fault tolerant. This is achieved by the same techniques as for application programs. Each of the replicated copies of the executive will use data from itself and the other copies. In the event of errors in one of the executives, the other copies will not use the data computed by it, thereby keeping their results valid. The correctly functioning copies will initiate a new copy of the executive in another processor (which will entail

* This requires availability of a sufficient number of the various units (processors, memories, busses).

copying the program to that processor) and will signal the malfunctioning processor to discontinue processing the executive. In addition on inspection of the data in the correct copies of the executive all processors will cease referencing the data in the incorrect copy, thereby preventing a system breakdown in the event that the malfunctioning processor continues processing the executive even though requested to discontinue.

The fault-tolerant procedures outlined above can be summarized as follows:

- Given at least triple replication, all single faults can be tolerated, and all uncorrelated double faults can be detected.
- Given greater resources (memories, busses, and processors), multiple uncorrelated or correlated faults can be tolerated.

It is expected that, in the event of a permanent fault detected, a unit will be relieved of any active part in subsequent calculation. Therefore, the capacity of the system will be reduced; however, until a large fraction of the system is faulty, the fault-tolerance procedures can be continued without jeopardy. It is expected that with the use of LSI circuitry, the different units will be replicated manyfold (e.g., 10 of each unit); for less critical applications, fewer units will suffice. The removal of faulty units will be accomplished by allocating them to null tasks in the case of processors, and by not referencing them in the case of memories. The overall effect of these strategies is to achieve a graceful degradation either of computer capacity or fault tolerance, whichever is desired in the particular application.

CONCLUSIONS

A system architecture has been presented that achieves great flexibility in fault-tolerance procedures. The salient points of the design objectives that are achieved are:

- Fault tolerance can be varied so that for some tasks it can be arbitrarily high, using suitable replication and reconfiguration strategies; for other tasks, the fault tolerance can be less.
- No special design requirements are placed on the processing units or memories, thereby enabling different designs to achieve different computer power.

The basic feature of the system is that high level fault detection, avoidance, and correction functions are achieved by software procedures rather than by special hardware. The increased computer load caused by the software fault tolerant techniques has not been assessed fully at this time, but it is expected to represent a reasonable cost for the benefits gained.

The system is currently in the design stage with many problems still to be solved. Some of these problems are also present in other fault-tolerant architectures. Typical among these problems is that of finding ways to fragment the memory so that only a part rather than a whole memory unit needs to be reconfigured. Another problem concerns finding methods of checking units of the system that are not regularly used. Reconfiguration software (or hardware in other architectures) will only be invoked infrequently and may itself have been subject to damage during its period of quiescence. In the system as described, this reconfiguration is carried out by a program whose correctness could be verified from time to time by inspection carried out by a program that reads and compares the multiple copies.

REFERENCES

1 A AVIZIENIS
Design methods for fault-tolerant navigation computers
Technical Report 32-1409 Jet Propulsion Laboratory
Pasadena California October 1969

2 A AVIZIENIS et al
The STAR (Self-Testing and Repairing) computer—An investigation of the theory and practice of fault-tolerant computer design
IEEE Trans Vol C-20 No 11 pp 1312-1321 November 1971

3 A AVIZIENIS
Arithmetic error codes: Cost and effectiveness studies for application in digital system design
Proceedings of Symposium on Fault Tolerant Computing
Pasadena California March 1971

4 W G BOURICIUS et al
Investigations in the design of an automatically repaired computer
Digest of the First Annual IEEE Computer Conference
Chicago Illinois September 1967

5 W C CARTER W G BOURICIUS
A survey of fault tolerant computer architecture and its evaluation
Computer Vol 4 No 1 pp 9-16 January 1971

6 W C CARTER et al
Logic design for dynamic and interactive recovery
Proceeding of Symposium on Fault Tolerant Computing
Pasadena California March 1971

7 W C CARTER P R SCHNEIDER
Design of dynamically checked computers
Proceedings of IFIPS 1968 Congress
Edinburgh Scotland August 1968

8 R S ENTNER
Presentation of advanced avionic digital computer baseline definition

Naval Air Systems Command Washington D C September 1969

9 J GOLDBERG K N LEVITT R A SHORT
Techniques for the realization of ultra-reliable spaceborne computers
Final Report Phase 1 Contract NAS12-33 SRI Project 5580 Stanford Research Institute Menlo Park California September 1966

10 J GOLDBERG et al
Techniques for the realization of ultra-reliable spaceborne computers
Final Report Contract NAS12-33 SRI Project 5580 Stanford Research Institute Menlo Park California June 1969

11 A L HOPKINS JR
A fault tolerant information processing concept for space vehicles
IEEE Trans Vol C-20 No 11 pp 1394-1403 November 1971

12 L J KOCZELA
A three-failure-tolerant computer system
IEEE Trans Vol C-20 No 11 pp 1389-1393 November 1971

13 G Y WANG
An in-house experimental aerospace multiprocessor—EXAM
ERC Memo KC-T-031 NASA Electronics Research Center Cambridge Massachusetts September 1967

14 G Y WANG
System design of a multiprocessor organization
Memorandum RC-T−179 NASA Electronics Research Center Cambridge Massachusetts 1969

TRIDENT—A new maintenance weapon

by R. M. FITZSIMONS

IBM Maintenance Technology Center
Research Triangle Park, North Carolina

Everyone is familiar with maintenance. It is a necessary requirement for almost everything we have, from spacecraft and automobiles, to the heels on your shoes. The maintenance that this paper is concerned with is that of business machines and their associated products. These devices are the means to the end required for your business and other endeavors.

Business machine maintenance in the recent decade has become more and more important to the user, the manufacturer, and of course the maintainer. For example, this is a paper on business machine maintenance. A few years ago this subject would have had a difficult time even being a sub-topic during your coffee break. Another example is the IBM facility in Raleigh, N.C. which treats maintenance as a technology in itself.

In spite of this new found attention, business machine maintenance is little different in principle than any other maintenance service. There are two principal factors affecting the maintenance task. The first, and the most familiar one is that of the product to be maintained. Product maintenance requirements are based on what the device is, how hard we use it, how much we pay for it, how well we treat it, what it is made of, who made it, etc. This is the maintenance factor that is intrinsic to the product itself. Certainly, the service component of the product has been with it from its earliest design days. Within IBM, this principle of service is very well covered by the maintenance function being involved early in the product development cycle. Examples of this type of maintenance planning are familiar to you in the form of service aids such as the flight engineer's console on a commercial jet liner, the oil pressure light on your automobile, or the stop on error switch on various components of your business machine.

The *other* key factor of service is the maintenance delivery system. This is a facet of service that is always present but very rarely discussed. When examined closely, this principle of service shows that maintenance is also a classical study in logistics. Whether discussing the spacecraft or the business machine, the problem is getting the *right man* to the *right place*, with the *right data* and the *right part*, at the *right time*. There are five "rights" to that preceding equation and all of them must fit "right on" for a successful maintenance task.

In the last ten years, things have been changing. Up until then, three of the "rights" were variable and two of them were fixed. For example, the three variable "rights" were and still are, the *right man*, the *right data* and the *right part*. The fixed "rights" were the *right place* and the *right time*.

The equipment user of the past always knew where the right place was. This has changed. In the complex teleprocessing systems of today, how often can anyone pick the *right place* on the first try?

The equipment user of the past always knew the right time. It probably was *immediately* because he could not use it. In the system of today, the user must now decide, *if* it is *immediately*. It could be immediately and someone fixes the machine while the user also runs it. It could be immediately if someone diagnoses the problem while the user runs it and it is fixed later. Or it may be the equipment problem will be diagnosed and fixed at a time that will be convenient for all concerned.

Thus, the logistics problem is not a simple one. It is not only very complex, but constantly growing in complexity. Not only is the complexity of the logistics problem significant, but so is the magnitude. The number of times this logistic system is exercised is quite impressive. The magnitude must be expressed in gee whiz numbers like, "It is my estimate that the maintenance logistics systems used to support the business machine industry in the United States is put to the test in excess of one million times per month, and that is probably being conservative".

The effectiveness of this logistic system in delivering the goods is also clearly recognized because, "Most of the time all of the previously identified five rights are done correctly." However, most of the time is no longer enough and "always," while it may never be, is certainly a rightful quest. It is only when the maintenance logistic system fails to deliver its first level of support that anyone acknowledges its existence and usually only then to condemn it. It must be repeated that a logistics system involves not only the maintainer, but the user. Remember, "where" and "when" are now variables.

There has always been a backup or second level support to the first level support system; it has many identities. However, it is familiar to most as "the specialist," the "factory man," or the "the guy coming in from out of town." In any case, at IBM this backup is part of an even more complex maintenance logistics system, because:

- There are fewer second level support people and they are more widely distributed.
- Second level support people face more difficult travel problems since they must go longer distances on unplanned activities.
- The second level support person may already be on another assignment and it would be difficult, if not impossible, to arrange an alternate support path.
- It is difficult if not impossible to have the second level support person bring with him extra data or information that he has collected at his home base.
- The time of day and the day of the week have a great bearing on how quickly and efficiently second level support can be brought to bear.

While these second level support problems are being resolved, the original problem has changed from the average or typical maintenance call to a *long call.* Second level support has a very high efficiency level. It can be said that "Most of the specialists fix most of the calls most of the time." This sequence can be repeated one or two more times depending on the problem and the service organization. However, each iteration and each sequence adds additional length to the call, especially as seen by the user.

The objective of any service organization is to collapse calls in the shortest possible time. This objective becomes a delicate balance between the maintenance characteristics of the unit and the maintenance logistics system of the service function. With new products, this balance must be reached as soon as possible. To do this, some of the items to be considered are the rate of a specialists learning curve, the initial distribution of the product, quality of the pre-shipment qualification, the initial stocking of the parts logistics system, engineering and sales change activity, and the complexity of operating system support.

There are solutions to the second level logistics problem. The easiest, of course, is to have all of the first level support be specialists. This is like saying, in music, that we need all composers instead of some composers and lots of arrangers. Maintenance does not require a composer or specialist all of the time, but when it needs one, the need is immediate.

The potential value added to a service organization by specialists is very great, and must be well controlled or the resource is wasted. Popular notion is that if a specialist could take or assist on every call, there would be a tremendous reduction in the length of calls. This just is not so. As stated before, during the first level of support, most of the men fix most of the calls most of the time. There is no value added by a specialist if the on-site maintenance package has already isolated the problem to a Field Replaceable Unit. There is no value added by the specialist if the tape drive motor is burned out and the only decision that can be made is to replace it.

The specialist must be used only on those calls where he can make use of his ability. The expense of a second and third support level is great and the service function must always balance the need against the cost. Otherwise, too much support exists when not needed, or not enough support when needed.

So, the proposition, which was to use only specialists, was not really a good answer. What seems a better answer is to build a logistics system that not only insures the right part is available, but that the right specialist and the right data is available at the right time. This new logistics system must function to the same or higher degree of accuracy than that which is built into a parts system. Parts logistics systems have been with us for some time; human and data logistics systems are something else.

One thing is for sure. One system is tough to control, two are mightly difficult, three may be nearly impossible. There is a great deal to look at when attempting to define such a complex system. However, as in any such effort, some part of the system will be completely new, but most of it will be built on previous experiences or facilities.

As a normal first step in system design, there must be a name for the system. For the purpose of this paper, the name "TRIDENT" was chosen. Since there are three major sections to this system, it seemed fair to name it after the three-pronged spear of Neptune, God of the Sea. Since "TRIDENT" must also be an acronym for something, let's say it stands for **TRI**ple **DE**fender of **N**ew **T**echnologies.

In building "TRIDENT," we can use existing parts logistics techniques. The other two delivery systems related to specialists and data are going to be the tough ones. The next easiest system to build seems to be the data system. However, very little is known about technical data. For example, what is its life cycle? More important, what is its half life cycle by user? What are the different index requirements required by the developer and the maintainer? How can one identify which data "must know" versus "nice to know."

The thing to do is get started and create a data system that will not only produce results, but will also provide the experience base and information necessary to complete the second step of "TRIDENT." In building this part of the system there is another "given" parameter that can be used. The first level support team does fix most of the problems most of the time with data that are already on site. The new logistics system will not try to replace this data, but will attempt to supplement it when necessary.

The goal, then, is to be able to deliver data when the first level of support needs it. Initially, to whom is it given? Enough is known about the distribution problems of data to decide to deliver it to those who will always be involved on the tough calls—the second level support team. In this way, early costs are reduced while getting the best level of return on investment. To send the data out to everyone is of great cost, especially when not all can make use of it, therefore, give it only to those few who have a great need.

What does this data look like? Most of it starts as precautionary or preventive information. It says, "Warning, safety change. Make the following modifications immediately." Another message may be, "Do not put this engineering change on machine Type A with Feature X unless you do the following." Or, "If you experience intermittent problems on Function J, check the following items." And even, "We are having trouble keeping Item Q from wearing. If you have this problem, call XYZ immediately."

There will be a lot of this type of information, therefore, good abstracting techniques will help reduce user reading time. This enables the user to read the text only if the abstracts interest him. Deliver the data daily to the users home base, also classify it in some way so that he has only to look at data concerned with his specialty. This creates a well informed specialist who has a good idea of the national picture of his product. You have boosted the learning curve by exposing maintenance personnel to the national picture of problems on his machine or specialty.

Now, the second level support man is getting armed on a daily basis with new information. He is better able to render on-site assistance. At least, he has a pretty good idea if someone else has either solved the problem or is already working on the solution to it. But there is still a difficulty here. When dispatching a specialist to an on-site call, he is moved away from his new found data base. He is now running on memory again. Agreed, this is better than it was before, but maybe there is also an even better way.

It does not take long to find out what is really wanted for a second level support technical data base. Output is wanted on a graphics device, with hard copy as an alternative. The user must be able to scan read large quantities of data to find what is wanted. All of the data scanned must relate to the problem. Also, the data that can solve a problem may be found under many headings in many different places. It may also be hidden in the text of data that is not even indexed under the required specialty or machine type. The technical data base must be searched not only by title and abstract, but also it must be searchable by each word in the *text*. The search must be fast. Certainly the search to identify what items must be reviewed should be consistently done in under ten seconds. Also, the stepping or paging through of the data while scan reading is in process must certainly consistently be achieved in under three or four seconds.

Another item that will set our TRIDENT design is the advantage of keeping the data base with the specialist. There are two ways to do this. One is to insure availability of graphics devices having access to the data base wherever the specialist is. The other is to keep the specialist and his data base access device fixed and move the difficult calls to him.

Neither alternative is easy to do but the second has the most going for it. If the call can be moved to a specialist, a major part of the third logistics problem is solved. Concentrations of second level support personnel can now be made. Skill backup and total coverage around the clock can be provided. Second level travel time can now be used for problem solving time, thereby increasing specialist efficiency. A problem solving environment can be maintained at all times. Also, all of this can be done without making any major technology breakthrough.

A call can be moved to a specialist in two mays. The most familiar is by voice connection. This one makes excellent use of the specialist's experience and his second level support data base. The first level of support man can identify the problem. The verbal exchange will soon identify a search argument for the data base. Answers will be developed and hopefully a high hit rate of problem solving capability will be achieved.

If the voice sequence fails, there is one more way of moving the call to the second level of support. That is via a teleprocessing connection between the equipment experiencing the problem and the second level support location. But this is not applicable to all types of equipment. Initially, where is a good starting point? The best place is with teleprocessing devices that can use the switched telephone network. Where else? Certainly, switched network teleprocessing capability can be added to other business machine equipment so that it can be manipulated by a specialist from a remote location. Some of the considerations that go into the selection of the devices that will have this new type of support are as follows:

- One connection must allow access to more than one unit or system to reduce cost.
- The device that has this connection must be capable of running an on-site maintenance package, even while the equipment that it is running on, is malfunctioning.
- The connection must be able to operate in any one of the maintenance environments selected by the user—concurrent, or dedicated. It must also offer significant growth advantages for new uses to insure it can make use of new applications without major new costs.

The choice is not too difficult, based on the following reasoning:

- The connection should be to a central processor so that more than one device can be supported by the single connection.
- The connection should be capable of connecting to multiple processors if they are in a single location.
- The connection should be to a large enough processor so that a sophisticated on-site maintenance package can be used.
- The connection should be into processor complexes that offer both maintenance environments.
- The connection should be into processor complexes that will offer new applications growth.

In summarizing all of these items, what is really desired is to connect into computer based systems that will insure long term usage.

Having completed the general design parameters of the third prong on our "TRIDENT" project, it is known that it will not be 100 percent effective in solving the second level personnel logistics problem. Dispatching men to the problem site to resolve some of the tough ones is still needed but every experience both good and bad will be valuable. As was pointed out in the beginning of the paper there are two principal factors to maintenance, the intrinsic maintenance factor and the logistics factor. The business machine can be maintained without a "TRIDENT" system. The mission of the "TRIDENT" system is to increase the effectiveness of the maintenance delivery system and reduce the length of the call as seen by the business machine user.

IBM has a "TRIDENT" in place today. In the Field Engineering Division of IBM it is known as the Parts Inventory Management System, the Field Support System, and a Remote Maintenance Support System made up of the Teleprocessing Test Center and RETAIN/370.

The Field Engineering Division's **P**arts **I**nventory **M**anagement **S**ystem (PIMS) is based in IBM's Distribution Center in Mechanicsburg, Pennsylvania. PIMS uses a System/360 Model 65 teleprocessing system to maintain both the parts flow and the administrative data required to insure a highly effective maintenance parts posture. This system makes use of the IBM corporate teleprocessing network as its prime branch office communications path.

The Field Engineering Division's **F**ield **S**upport **S**ystem (FSS) operates out of the Division's Management Information System Center in Sterling Forest, New York. FSS uses a System/360 Model 75 connected to hard copy terminals located in Division Headquarters in White Plains, New York and in branch offices and plant sites. FSS hosts two major applications. The first is a technical information distribution system. This is the application that keeps field specialists aware, on a daily basis, of the latest technical information in his area of expertise. The technical information is provided in both abstract and text form. After reviewing technical abstracts the specialist can retrieve full text information as required.

The second application on the Field Support System is called the Field Instruction System (Figure 1). FIS is a coast-to-coast computer-based instruction system that provides self-study training for IBM customer engineers with the Field Engineering Division, which installs and services information handling systems and equipment.

The availability of computer-assisted instruction in every branch office has two benefits:

- It reduces the time that customer engineers otherwise would have to spend away from their office while training at an education center, thereby increasing the availability of key resources at the point of application.
- And consequently reduces the cost of education while achieving the course objectives.

The branching capability and storage capacity of the system permit the student to:

- Master a new topic at a personalized pace.
- Skip over topics already mastered through experience or previous training.
- Receive help upon request from sequences prepared to clarify difficult points in the course.
- Test his new knowledge.

The student's interaction with the system and its flexibility in meeting different student needs increases his acceptance of this form of training, and prepares him to perform his service skills effectively.

Today, as one customer engineer studies through the Field Instruction System, he will be sharing the computer with classmates from Maine to California. The other students could be studying the same course or any of the other courses stored in the computer.

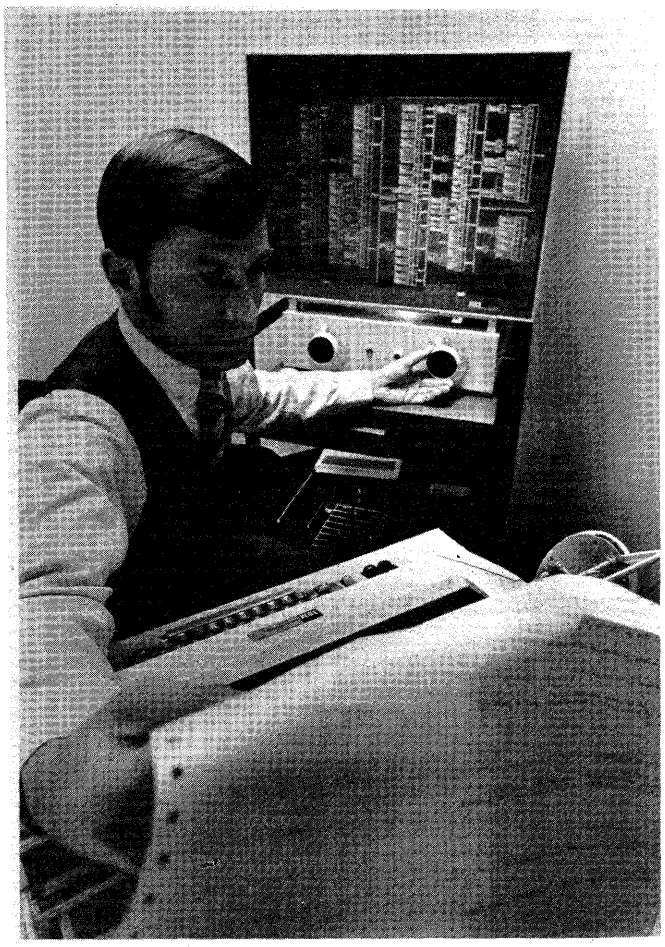


Figure 1—*Field Instruction System*—A coast-to-coast computer-based instruction system provides self-study training for IBM customer engineers servicing data processing equipment.

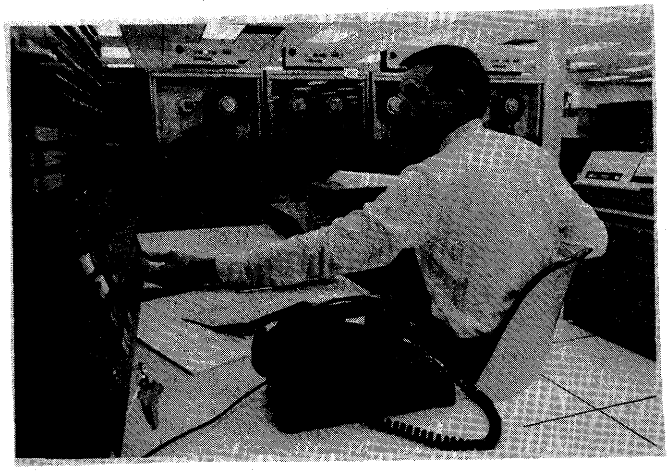


Figure 2—*Retain/370*—An IBM customer engineer has immediate access to a remote source of maintenance information for fast service to IBM System/370 customers through RETAIN/370 (Remote Technical Assistance and Information Network/370). It helps minimize the duration of interruptions to customer operations due to problems with IBM equipment or programming. An IBM customer engineer can obtain information from a specialist at a strategically located Field Technical Support Center, thus saving time by reducing the need for the specialist to travel to the customer site.

The courses, as written and programmed into the computer, simulate the interaction which might take place in a conventional classroom between a student and his instructors, and supplement education activities at the division's education centers throughout the nation.

IBM customer engineers have immediate access to a remote source of maintenance information called RETAIN/370, (Remote Technical Assistance and Information Network/370) which combines technical support with comprehensive computer files of maintenance information. It helps minimize the duration of interruptions to customer operations due to problems with IBM equipment or programming. It saves time by reducing the need for specialists to travel to the customer site, and by enabling customer engineers to quickly obtain information on a wide variety of problems and solutions.

RETAIN/370 is used by the IBM customer engineer when a problem cannot be defined or resolved with on-site diagnostic techniques within a short period of time (Figure 2). To access RETAIN/370, the customer engineer contacts his technical support center via the dial-up network and describes the problem. The specialist then searches the data base for fixes that relate to similar problems, or together, the customer engineer and the specialist may use the data link to run maintenance programs. The output can be dis-

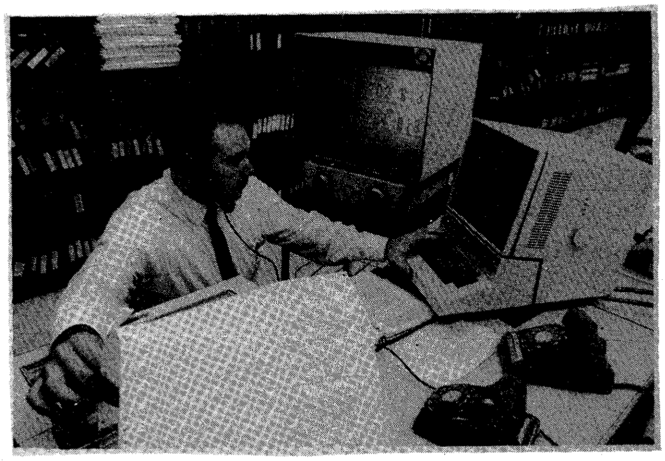


Figure 3—*Retain/370*—An IBM service specialist at a strategically located Field Technical Support Center uses RETAIN/370 (Remote Technical Assistance and Information Network/370), a teleprocessing network, and the power of a computer in Raleigh, N.C., to diagnose malfunctions remotely. The specialist, working with the customer engineer at a customer location, can search a data base for solutions, or together, the customer engineer and the specialist may use a data link to run maintenance programs. The output can be displayed both on the customer's System/370 and on the specialist's display termainl.

played both on the customer's System/370 console and on the IBM 2915 display terminals in the Division's Technical Support Centers in New York, Chicago, and Los Angeles (Figure 3). Problems and corrections discovered are fed back to the RETAIN/370 data base for future reference. In this way, resolutions of new problems can be made available within minutes to IBM customer engineers working on System/370's throughout the country.

The system consists of three major components: the customer installation, the Field Technical Support Centers and the RETAIN/370 support center in Raleigh. The focal point of the system, though, is the customer installation and the customer engineer on location who performs the actual maintenance.

The *customer account* includes the customer engineer, the customer's System/370, diagnostics and other standard Field Engineering Division maintenance tools such as oscilloscopes, meters, maintenance manuals, etc. Also included at the customer account is a specially designed IBM Field Engineering tool, the 2955 data adapter unit. This unit provides a data link between the customer's System/370 and the RETAIN/370 system. This data adapter allows a specialist, together with the on-site customer engineer, to initiate and control maintenance tests in the customer system, and view transmitted test results and other maintenance data remotely.

The specialist in a *Field Technical Support Center* is equipped with a display terminal, a printer, a data phone and a microfiche viewer, all of which provide rapid access to maintenance information. The support centers are also equipped with machine logic diagrams and reference manuals, program listings and other normal maintenance publications.

The third major component of the system is the *RETAIN/370 Center* in Raleigh, North Carolina. It utilizes a System/360 Model 65 with teleprocessing links to Field Technical Support Centers as well as to Domestic IBM laboratory sites. A centralized data base and a data link are the heart of RETAIN/370.

Included in the data base are such standard service aids as:

- An index to service publications covering theory of operation and maintenance manuals.
- An Engineering Change Announcement index.
- An index of service aids providing reference to microfiche text and service aid abstracts.
- An index to programming documentation.

The data base also contains special information files, such as:

- Symptom/Fix file, a temporary data storage built from experience data provided by customer engineers, and support personnel.
- Incident Log, a running log of statistical and technical information developed during the resolution of problems.
- Specialist Log, used by each specialist for notes of technical interest.

RETAIN/370 provides two data searching and retrieval methods: one interpretive, and the other incremental.

The interpretive method is used when the existence or location of specific maintenance information is not known. With this method, the data base can be searched by using a series of key words entered by the specialist from information provided by the customer engineer.

The incremental retrieval method uses progressive index levels, such as the master index, which lists all machines by unit type. When a unit type is entered, a sub-index lists related information categories. A selection from this page produces a list of abstracts on the selected category. When an abstract is selected the full text of the selected record is shown.

The data link feature of RETAIN/370 transmits the results of diagnostic programs operating in the customer's System/370 to the Field Technical Support Center. Then the customer engineer and specialist can examine the same information to further diagnose the customer's problem.

After obtaining customer approval to use the data link, the customer engineer uses the 2955 data adapter unit and changes from voice to data mode. The results of the diagnostic programs can be simultaneously reviewed by the customer engineer and the specialist at the Technical Support Center. Where security is a concern, all data transmitted can be stored or printed for customer inspection.

The RETAIN/370 support system is available 24 hours a day, 7 days a week.

IBM customer engineers servicing teleprocessing equipment anywhere in the country can telephone the Teleprocessing Test Center in Raleigh, North Carolina. They use on-line diagnostic tests similar to ones that are available to run in the customer's system to track down difficulties in teleprocessing terminals and other communications equipment. The customer engineer can also confer with a test center specialist.

In operation since October 1969, the center has proved to be a fast, efficient service aid. It provides the customer engineer with remote diagnostic data and verifies IBM teleprocessing equipment operation—usually within minutes—without interrupting the customer's system. In effect the Teleprocessing Test Center is a substitute host system for a teleprocessing terminal.

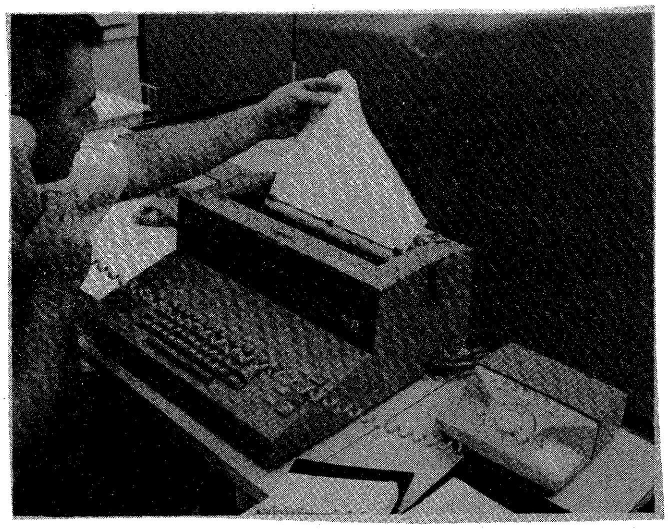


Figure 4—*Teleprocessing test center*—A customer engineer servicing IBM teleprocessing equipment anywhere in the country can telephone IBM's Teleprocessing Test Center in Raleigh, N.C. at any time. He uses diagnostic tests to track down difficulties in teleprocessing terminals and other communications equipment. He also can confer with a test center specialist in Raleigh for further assistance.

Figure 5—*Teleprocessing test center*—A service specialist at IBM's Teleprocessing Test Center in Raleigh, N.C., confers with an IBM customer engineer at a customer location to track down difficulties in IBM teleprocessing equipment.

Using the regular telephone network, the customer engineer calls the test center from the customer's teleprocessing machine (Figure 4). After dialing and signing on with a special number the center's computer runs a general diagnostic test. A specific test tailored to a particular problem also can be requested. The diagnostics test the customer's terminal and control equipment with a series of exercises. The results of the exercises are sent to the customer engineer for his analysis.

If during the testing, the customer engineer needs more information, he can press the "talk" button on the data set and confer with a test center specialist. This action flashes a signal to the display terminal alerting a specialist that assistance is required. From his display terminal, the TP specialist can monitor the diagnostic tests being run by the customer engineer (Figure 5). The experienced specialist can often recognize the difference, for example, between a properly operating terminal and one that isn't, by merely listening to the signal.

Besides handling trouble calls, the test center reduces installation time of new teleprocessing systems. As each machine arrives, the customer engineer can install and check it, whether or not other parts of the system are installed.

A small portable diagnostic device also allows IBM customer engineers to test IBM's teleprocessing equipment that do not normally attach to the switched network.

This portable device, the 1200 Baud Terminal Diagnostic Analyzer and Tester (1200 TDAT) is especially useful to teleprocessing installations using dedicated lines because it makes available the facilities of the

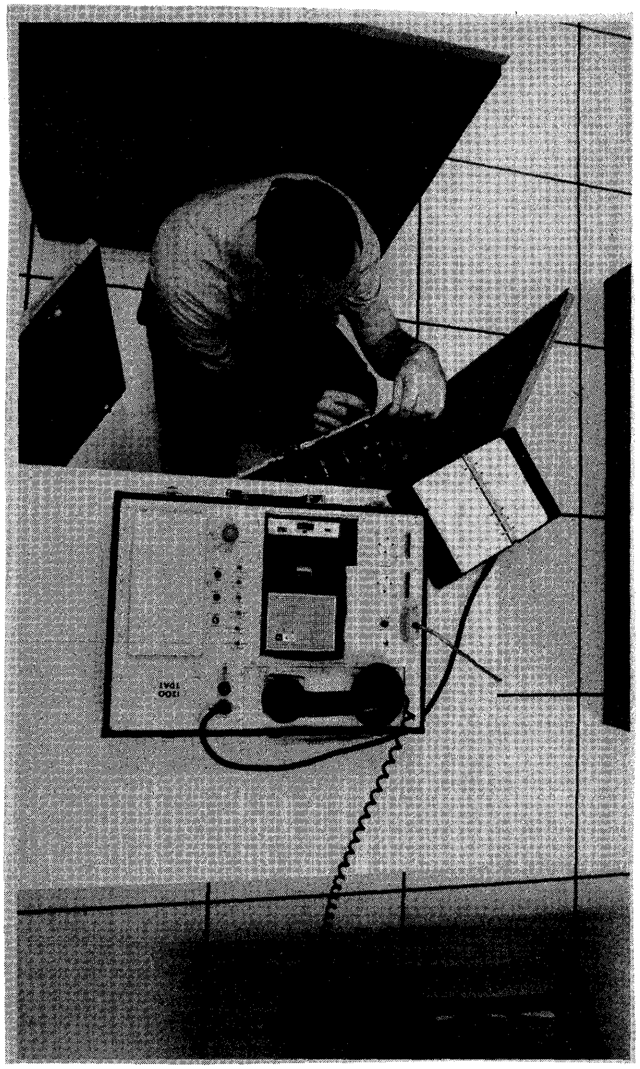


Figure 6—*1200 Baud terminal diagnostic analyzer and tester*—A small portable diagnostic device allows IBM customer engineers to test virtually all of IBM's teleprocessing equipment. This portable device, a 1200 Baud Terminal Diagnostic Analyzer and Tester, enables customer engineers across the country to test teleprocessing equipment via telephone through IBM's Teleprocessing Test Center in Raleigh, N.C.

Teleprocessing Test Center. Previously, only teleprocessing installations with access to the telephone dial network could connect to the TP Test Center (Figure 6).

TDAT features an acoustic coupler, tape recorder, and a modem that allows connection through leased, or dedicated telephone lines, from a teleprocessing terminal at one location to another at a remote location.

Besides the acoustic coupler feature, the portable testing instrument can do the following:

- Simulate a data set, another teleprocessing terminal or even a remote central computer by being able to play recorded data into a terminal.
- Monitor a teleprocessing system, and when an error occurs, stop and store data leading up to, and including, the failure for analysis.

By recording data as it is transmitted from or to a terminal, the Terminal Diagnostic Analyzer and Tester in effect verifies transmitted data. The tape recorder also can be used as an exerciser to service a terminal with a malfunction. In this way, a customer's central processing unit need not be tied up exercising the terminal.

IBM has demonstrated that the power of data processing equipment can be successfully applied to solving the logistics problems of maintenance.

In conclusion, it can be stated that IBM, and therefore IBM's customers, have benefited from the Field Engineering Division's endeavors based on two concepts. The first is the concept that maintenance is a technology in and of itself and, therefore, just as with other technologies, is amenable to investment for innovation.

The second item is the concept that successful maintenance involves the effective solution of a total system problem in logistics. By clearly defining the problem, the application of innovative effort is being accomplished in an optimal fashion.

Computer system maintainability at the Lawrence Livermore Laboratory*

by JOHN M. BURK and J. EDWARD SCHOONOVER

University of California
Livermore, California

INTRODUCTION

Since LLL's computer complex, or network, is in operation 24 hours a day, 7 days a week, maintenance procedures and controls have been or are in the process of being developed which minimize disruption of user service. The challenge of developing such tools and procedures is intensified by the diversity of hardware within LLL's system—CDC, IBM, DEC, Ampex, Lockheed, General Precision,* and LLL—and by the number and type of users on-site—1,000-plus scientific and administrative users. Although designed for a time-sharing system (designated at LLL as the Octopus), many of the tools and procedures apply to a stand-alone system as well since the integrity of each user (host) computer to function as an independent entity has been preserved. Tools and procedures described include on-line and off-line diagnostic software. In addition, fail-soft procedures (recovery procedures effecting minimal system interruption) developed at LLL are described. In conclusion, the diagnostic tools and procedures are evaluated and findings from samplings of system availability are presented.

This paper presents not a theoretical approach to the problem of computer system maintainability, but rather the evolutionary techniques extant at Lawrence Livermore Laboratory.

OVERVIEW OF LLL'S COMPUTER NETWORK AND ADMINISTRATIVE POLICIES

Description of network hardware

LLL's computer network presently has five user, or host, computers; namely, three CDC 7600's (Serial Nos. 1, 2, and 17), a CDC STAR-100 (Serial No. 1), and a CDC 6600 (Serial No. 13). A letter identification (R, S, T, A, and L) has been used to designate each machine (Figure 1).

The CDC 6600 has a 128K-word memory, ten PPU's (peripheral processing units), three disks with approximately 1.3 billion bits, eight tape drives, a card reader, a printer, a punch, and a DD80 35mm microfilm recorder and display scope.

Each CDC 7600 has a 64K SCM (Small-Core Memory), 512K LCM (Large-Core Memory), ten PPU's, two disks with approximately 10-billion bits, a 160-million bit drum, eight tape drives, a card reader, and a printer.

The CDC STAR-100 has a 512K-word core memory, five input/output (I/O) stations, two paging drums (approximately three times core memory), two disks (7600 equivalent), four 9-track tape drives, a card reader, and an on-line printer.

The control, or hub, computer consists of two DEC PDP-10 processors and their shared 256K-word memory (Figure 2). It is directly connected to the host computers by high-speed (12 MHz) interfaces to transport data between the host computers and the shared data base which consists of a trillion-bit IBM Photostore, a 3.2 billion-bit IBM data cell, and eight 707-million-bit CDC 844 disk packs. The hub computer also has dedicated a 880-million-bit General Precision Librascope disk which is used as an intermediate storage and buffering device. A PDP-10 processor also controls the TMDS (Television Monitor Display System) which via a 16-channel, 128-position switch connects 128 television monitors, each of which can have additional monitors serially attached. A color capability also has been implemented.

Four DEC PDP-8's serve as concentrators and control the Teletype (TTY) sub-network. Each PDP-8 may have 128 interactive teletype terminals attached and may be connected to two host computers. A PDP-8

* Work performed under the auspices of the U.S. Atomic Energy Commission.

* Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Atomic Energy Commission to the exclusion of others that may be suitable.

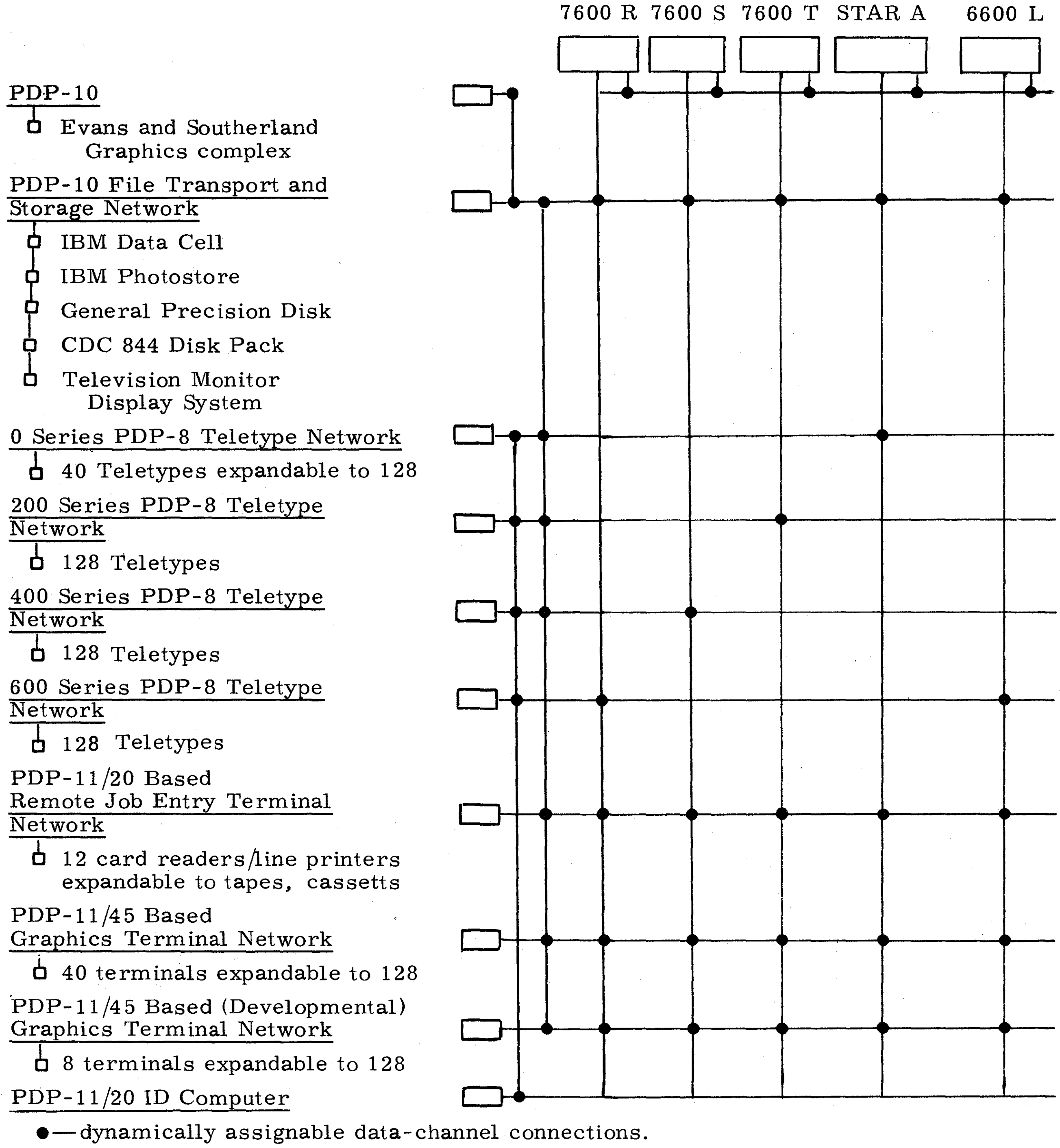

7600 R
7600 S
7600 T
STAR A
6600 L
PDP-10
Evans and Southerland Graphics complex
PDP-10 File Transport and Storage Network
IBM Data Cell
IBM Photostore
General Precision Disk
CDC 844 Disk Pack
Television Monitor Display System
0 Series PDP-8 Teletype Network
40 Teletypes expandable to 128
200 Series PDP-8 Teletype Network
128 Teletypes
400 Series PDP-8 Teletype Network
128 Teletypes
600 Series PDP-8 Teletype Network
128 Teletypes
PDP-11/20 Based Remote Job Entry Terminal Network
12 card readers/line printers expandable to tapes, cassetts
PDP-11/45 Based Graphics Terminal Network
40 terminals expandable to 128
PDP-11/45 Based (Developmental) Graphics Terminal Network
8 terminals expandable to 128
PDP-11/20 ID Computer
●—dynamically assignable data-channel connections.

Figure 1—Octopus distributed network

to PDP-8 connection allows the teletype sub-network to function independently of the PDP-10 hub computer. A PDP-11/20 ID computer is connected to the PDP-8's; its primary duty is to act upon and control individual user accessibility within our security framework (Figure 3).

Two Princeton Electronics graphic terminal sub-networks are used for remote visual graphical interactivity. They have a 256-expended-character-set capability.

An additional sub-network of RJET's (Remote Job Entry Terminals) provide for I/O at remote locations of the Laboratory. At present, each terminal consists of a

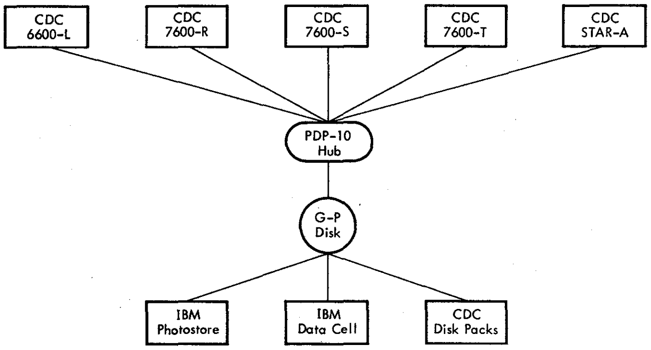


Figure 2—File transport and shared data base network

TTY, 600-lpm printer, and 400-cpm card reader; however, they can be expanded to include magnetic tape and tape cassettes.

A user mode within the hub computer system allows the use of one PDP-10 processor to drive the Evans and Sutherland, LDS-1 computer and its associated interactive graphics terminals.

Administrative policies and procedures

User access accountability

All network accesses, whether it be day or night, individual user or computer operator, are via TTY remote terminals. An identification message must be transmitted which identifies the host computer being addressed, the user, the user's division, and, if required, security level accessibility. Additional operator and division user numbers are required during production periods under operator control. (Job mix, priority, interactivity, and maintenance procedures are controllable by the operator.)

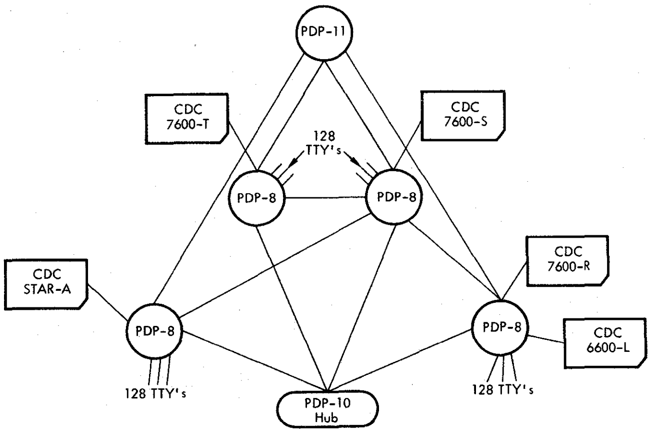


Figure 3—The teletypewriter sub-network

The executive systems—designated STAR on the CDC STAR-100, FROST on the CDC 6600's, FLOE on the CDC 7600's, and HYDRA on the PDP-10—verify the ID messages and establish appropriate linkage. In addition, the executive systems verify time allocation by machine per day, night, and weekend; authorized users within each divisional account; and the percentage level each user may draw upon its division's total time allocation for each period.

LLL's executive systems are written and maintained in-house. While it may seem that much effort is spent "inventing the wheel," at least that wheel precisely fits the vehicle for which it is intended. The time delays normally associated with adding new system features or fixing old ones are minimal, and the constraint of compatibility with the rest of the world does not exist other than at compiler and assembler levels. The ability to tailor-make an executive system architecture has facilitated the implementation of LLL's maintenance and fail-soft procedures.

Dissemination of network performance to users

All TMDS terminals, when otherwise not in use, display a dynamic system status which is continually up-dated by automatic system messages and by operator-initiated information messages. Automatically, the PDP-10 (hub computer) periodically pulses each of the components attached to it and displays their status on the TMDS monitors. For instance, if a user is running direct from a TTY to a host computer via a live and healthy PDP-8 and if the interface connecting that PDP-8 to the PDP-10 is not functioning, that linkage would be reported as failing. Each "Operator Information" (OP INFO) message initiated by the operator or automatically by the system is placed in a buffer and sent to all TTY's, in addition to being displayed on the TMDS. TTY messages forewarning the user of system interruption, tape backlog, and system dead starts, for example, are helpful in reducing user frustration (sometimes).

Another communication medium used is the "Octogram," a daily news release which keeps the user up-to-date on day-by-day activity. The "Octopus, Communique" is a more detailed and permanent documentation medium sent to all computer users. These communiques describe system modifications, additional sub-routine or utility routine functions and other information of a permanent form prior to its release in a formal document or manual.

Bi-weekly C.I.E. (Computer Information Exchange) meetings enable further communication. On alternate weeks, Computation staff members meet with representatives from LLL's major computer user divisions and departments to discuss methods which, hopefully, will result in procedures that will satisfy their needs.

Fault analysis and liaison

First level investigation and determination of computer malfunctions is undertaken by the Systems Operations Section (SOS) of LLL's Computation Department. This section acts as a focal point and collection agency of facts and determines appropriate remedial action. SOS assists the operating staff, system programmers and the various engineering maintenance personnel (IBM—Data Cell/Photostore; CDC—Host Computer; LLL—Hub Complex) on an on-call, 24 hours-a-day basis.

In addition, an operator on each shift is appointed to keep in close touch with the Systems Operations Section, and each engineering group responsible to the time-sharing system also has an appointed liaison. Consequently, explicit formal channels exist for rapid communication.

MAINTENANCE TOOLS AND TECHNIQUES USED

Host computer diagnostic software

On-line software automatically scheduled

A subset of manufacturer's standard diagnostic software for the CDC 6600 and CDC 7600 is used to check functional units, memories, arithmetic precision, and random operand performance. Since these routines are automatically scheduled by the executive system and run as part of the normal operational job mix, they are subject to all the operating system idiosyncrasies of scheduling, loading, and timing and provide in a real-time sense a meaningful measure of hardware status.

The CDC 6600 routines run every 30 min for 20 sec each. The CDC 7600 routines run every 15 min for 1 sec each. Errors cause the offended routine to be restarted at twice its current time limit. The restart process has been programmed to continue until the error is no longer noted (error designated as intermittent) or machine is extensively diagnostically pre-empted (solid). Errors noted are sent to the operator's output TTY station, and pertinent memory dumps are hardcopied for the customer engineer. If the diagnostic failures are intermittent, maintenance decisions become a value judgment; that is, if the frequency of the error is low, immediate maintenance action may be deferred.

On-line remote execution software

On demand or desire, diagnostic software may be executed from remote stations without noticeably perturbing the operational job mix. This remote execution may be initiated by any user; however, it is usually done only by systems or maintenance personnel. An extensive open-ended job set is available which has been designed to exercise the mainframe and central processing unit (CPU) as well as peripheral hardware.

Off-line diagnostic software

Maintenance actions may require the suspension of all user services. If the suspension can be scheduled and does not involve write destruction of disk or memory system tables, the operational job mix can be saved to disk and automatically restarted after the maintenance action is complete. Manufacturer's standard diagnostic software is available.

Hub computer diagnostic software

On-line software automatically scheduled

The following diagnostic tests are automatically scheduled and executed by the executive system:

(1) Every 30 sec, 1 page of random data (512 36-bit words) is written to the G-P Librascope disk, read back, and compared. The disk controller hardware VERIFY function is also checked, and pertinent error comments are output to the operator's console TTY station.

(2) Every fifth data burst (maximum 64K 60-bit words) out of the hub computer to any device is read back and compared. Pertinent error comments are output to the operator's console TTY station.

(3) Every 30 sec the status ("hung" or "responding") of the IBM Data Cell is sampled. Pertinent error comments are output to the operator's console TTY station.

(4) Every time the IBM Photostore is referenced, its status ("up and available," "disconnected," "in recovery," etc.) is recorded on the TMDS, and pertinent error comments are output to the operator's console TTY station.

(5) Each time the TMDS display is updated (3.5 sec), the hub computer sends a message to each host computer and to each PDP-8. If there is no reply, this fact is noted by the hub computer. If

the device fails to reply three times in a row, the status is recorded on the TMDS as follows: "DOWN" for the nonresponding PDP-8 and "N/R" for the nonresponding host; date and time of status report are also included.

(6) Every error detected by the hub computer when reading or writing on the G-P disk, Data Cell, and Photostore is trapped and analyzed. A 15-line message, which includes the device, time of error, type of error, status of all control registers, and number of retries, is output to the engineer's TTY station. These printouts become a permanent log of all I/O errors detected by the hub computer. Persistent errors (catastrophies) invoke an automatic recovery procedure involving the hub computer executive system reload. Error comments are made, and three bells are sent to all TTY stations signalling the event. Every attempt is made to insure the integrity and automatic completion of the hub's job queue.

On-line remote execution software

Diagnostic routines designed to evaluate network components may be executed from a remote station by systems or maintenance personnel. These routines are time-shared in the hub computer.

(1) Specialized tests include:

- Photostore controller tests
- disk pack tests
- data acquisition system tests

Each tests the specified devices under simulated operation conditions since the device itself must be off-line. The routines send control messages, set or read status registers, and check data transmission using any of the various sub-channels available.

(2) Inter-machine tests
A complex of six routines is used to exercise and evaluate all possible communication paths within the file transport network. These routines determine the status of control functions, communication links, clocks, and interfaces between the hub computer and the designated host or remote terminal. (Interfaces include multiplexors, selectors, adapters, file channels, and line units.) When required for the test, a PDP-8 and a host PPU or remote terminal are dedicated to the diagnostic tests and unavailable to the operational network.

Off-line diagnostic software

In a stand-alone mode, an additional series of diagnostic programs exist which include operational tests for:

- console TTY
- tape reader/punch
- CPU instructions
- priority interrupt hardware
- data transfers
- memory protection and relocation
- processor timing
- I/O bus
- internal clock
- G-P disk—saturation
- memories—LLL, Ampex, Lockheed
- Data Cell

These tests are of two classes: (1) those which send control messages and verify responses, and (2) those which test data transmission paths by sending patterned data, having the device echo the data back via hardware control, and then comparing with the original data. All these tests can run in either a single-step or continuous mode.

Fail-soft procedures

Fail-soft procedures are considered to be those recovery processes which allow a resource flexibility and rapid automatic error recovery.

Commands enabling resource flexibility and sharing

Circumstances arise when it is desirable to have within the system an easy method of managing hardware resources. In developing this fail-soft capability, a comprehensive set of commands has evolved.

Commands available include:

P ALL HSP	Send all printer output to tape for off-line processing on the high-speed printer. The printer can now go down for maintenance.
D N MMM	N is a disk unit number 1 through 4. MMM is either "In" or "Out." This will allow or prevent, respectively, the

	creation of new files on disk N. If "Out" existing files remain accessible and disk N can be scheduled down for maintenance.
SP NNNNNN IN TEXT	Allows only privileged user number NNNNNN access to the host. All users' programs are saved on disk, and all attached TTY stations are logged out. The TEXT is sent to all users attempting to log in to the host.

For a complete list of commands refer to the Appendix.

These commands are initiated from the operator's input console TTY station and communicate with the executive system or with any active user's remote TTY. The commands provide a necessary resource flexibility in that no hardware device is permanently dedicated. Within the framework of these commands, the operator can provide back-up capabilities for I/O devices such as drums, on-line printers, and on-line punch. He may also elect to restrict the creation of all disk files to a specified disk unit or to prevent the usage of specific disks and/or tape units.

This management of tapes, disks, drum, printers, and other resources is desirable not only for back-up purposes, but also to make maintenance activities easier. The operator may take any specified device off-line for maintenance without disturbing the job queue or interrupting service to the active users. These devices may also be returned to service without interruption to the active user. In addition to the ability to manage peripheral hardware resources, there are commands that terminate the time-sharing process and restrict use of the system. This capability is particularly convenient when a suspect hardware failure develops that requires the host system to be dedicated to the task of error detection or machine maintenance. When this requirement exists, the operator disconnects all codes that are active in memory and requeues them on disk. The integrity of these codes is assured since there is an option within the command set to re-initialize the job queue. Similar techniques to manipulate devices attached to the hub computer are available to a privileged set of users.

Additional hardware flexibility is achieved by resource sharing: File transport channels can be used as secondary back-up routes for transmitting Teletype message packets when a PDP-8-to-host machine link is lost. Card readers and tape transports may also be shared by more than one host computer.

Dead Starts

The philosophy of dead starts has always been to minimize the consequences resulting from the dead start. The dead start options which have evolved represent LLL's progress in realizing that goal. Although circumstances requiring manual intervention and the consequent dead start vary, every effort is made to use the option inflicting the least hardship on the user. At the "softest" level, all disk files are preserved and disk queue jobs eligible for loading are automatically restarted. Unless the SP NNNNNN IN command was used before the machine failed, all jobs residing in memory are disconnected and removed from the operational job queue. At the "hardest" level all files are lost; all jobs are disconnected; and a recent tape copy of the public (permanent) files is restored to disk.

The dead start commands which follow are initiated from the engineer's console keyboard/display scope:

DS	Preserves all disk files and the disk queue. All in-memory jobs are disconnected.
DSD	Identical to DS plus a pertinent memory dump is hard copied for the system's programmers.
DSR	Preserves all private files and the disk job queue. All in-memory jobs are disconnected. Public files are restored.
DSU	Preserves most disk files; all jobs are disconnected. The disk file catalog (index) tables are restored from drum. Since the file index to drum save is a periodic function (2 min) some temporary (private) disk files may be lost.
DSN	Preserves public files. All private files are lost, and all jobs are disconnected.
DSB	All public and private files are lost, and all jobs are disconnected. Public files are restored.
	For the preceding dead start options, all executive system reloads are initiated from the drum.
DSC	Transfers the executive system from tape to drum; a DSC must be followed by the appropriate dead start option.

Automatic recovery for memory parity errors

An automatic recovery procedure for intermittent memory parity errors has been implemented within LLL's 7600 executive system (FLOE). This fail-soft technique requires no operator intervention, maximizes user availability, and allows for the prescheduling of memory maintenance actions. In the case of a memory error in a user code area, only that code is affected. All other jobs proceed normally and without interruption. The affected code is disconnected and removed from the operational job mix. Errors occurring in executive system memory areas require a reload (dead start) from drum of the affected system coding or tables. This procedure requires less than 1 min. The integrity of the operational job mix is subject to the particular dead start option, DS or DSU, instituted. The automatic recovery procedure does allow for a deferment of maintenance actions to a time period less visible to the user. This ability to schedule emergency maintenance periods has maximized system availability during prime usage periods.

Fault possibilities and dead start options are as follows:

Memory Error In	*Option Instituted*
Resident executive system coding	DS
Resident executive system tables	DSU
Resident user program	None*

The operating system determines the intermittency of the memory parity error by four-patterned reads and writes of the affected memory area(s). If the error does not reappear, automatic recovery is initiated and normal time-sharing resumes. A pertinent diagnostic comment describing the error is output to the customer engineer's TTY station and the operator's console TTY station. An entry is made in the event-file, and a fatal error status is returned to the offended user's program if the error occurred within a user's program.

It is a standard operating procedure to require emergency maintenance (EM) if the *same* recoverable parity error occurs twice within 2 hr. The SP NNNNNN IN command would be used to allow for the automatic restart of the operational job mix at the completion of the maintenance action.

Dynamic disk flaw tables

Before a CDC 7600 disk file is declared available for general use, a static table of flaws is generated. These flaws represent areas of the disk that contain hard faults (solid read parity). These data are incorporated into the appropriate operating system tables during a DSN or DSB dead start option and remain a permanent part of the systems flaw table data base. Flawed areas, one disk sector (512 60-bit words) in length, are not available for assignment to a program requesting disk space.

During normal time-sharing periods, the system flaw table data base is dynamically maintained. After thirty-two consecutive disk read failures, an entry is made in the file catalog index of the offended disk file, and an appropriate error status is returned to the program. No further system action is taken, and the disposition of the offended file is entirely under the control of the executing user program. When the program releases (destroys) the offended file, executive system action is required to ascertain the solidity of the flaw.

If the executive system cannot perform an error-free read of the disk sector containing the flaw, the flaw table data base is dynamically expanded. A diagnostic comment detailing the error is output to the operator's console display scope and the event-file. The dynamic flaw table entries are maintained over all dead starts not requiring the loss of private files (see dead start section).

Ideally, no static flaw tables need to be maintained; however, user frustration levels have been dramatically reduced by not requiring the continued re-discovery of known hard flaws. The transference of flaw entries from the dynamic to the static table is done periodically when it is determined an area of disk has indeed developed a hard fault.

Magnetic tape integrity

Insuring the integrity of magnetic tapes and associated tape transport hardware continues to be a major maintenance problem. User frustration levels reach all time highs when recorded information cannot be reliably retrieved. Since tapes seem destined to be with us for a considerable period of time, a major effort has been made to alleviate the problem.

All manufacturers suggested hardware modifications have been made. Vendor maintenance has been increased. All physical tapes are certified before release. All operators are educated in proper tape handling procedures. Tape transport heads and vacuum columns are cleaned periodically (once per hr).

Assuming perfect tapes and functional hardware, tape integrity is now assured. In a real world sense, however, extensive write recovery software procedures had to be implemented. The primary assumption is that if a tape can be written with no error indication, it can be read error free. Therefore, during the write operation, only bad parity records must be verified as having been rewritten correctly, i.e., the rewritten record and associated record gap and erased area must be error-free readable.

* The faulted user program is disconnected.

Utilizing this recovery has reduced our non-recoverable error rate to less than 0.01 percent from as high as 10 percent.

An extensive remote time-shared tape testing program is also available on the host computers. This test allows the simultaneous evaluation of tape transport hardware, software drivers, and physical tapes in the real-time environment.

Stand alone ability

The integrity of each host computer to function as an independent entity has been preserved. In the advent of a failure in the PDP-8/PDP-10 Teletype network, the ability to communicate with a host would be completely removed. While that portion of the host's operation job mix requiring no Teletype interaction would continue to run, no new jobs could be entered in the system.

To prevent the host from becoming "idle," a Teletype Simulator (TTYSIM) version of the Livermore time-sharing system can be loaded. This system relies entirely on a console keyboard/display unit as the interactive input and output media. Other than the fact that system operation is now completely operator controlled, no restrictions or limitations are imposed on the operational job mix.

Current preventive maintenance policy

If it were feasible to have total hardware redundancy, on-line maintenance would not be required since all component repair time would be off-line, and hence invisible to the user. This, however, is not the case; therefore, maintenance policies and procedures must be established which attempt to insure minimum network degradation while maximizing total system availability.

Two diametrically opposed maintenance policies have been periodically tried and systematically discarded: (1) schedule large amounts of maintenance, and (2) schedule no maintenance.

Scheduling extensive maintenance periods did not work. Not only was the device or component off-line and unavailable for extended periods, but faults requiring an unacceptable number of unscheduled maintenance periods continued to occur.

Scheduling no maintenance periods only compounded the unscheduled maintenance problem and indeed resulted in significantly decreasing overall total system availability. By not allowing any scheduled or preventive maintenance periods marginal logic cannot be detected and replaced, nor can maintenance actions designed to increase total system reliability be performed.

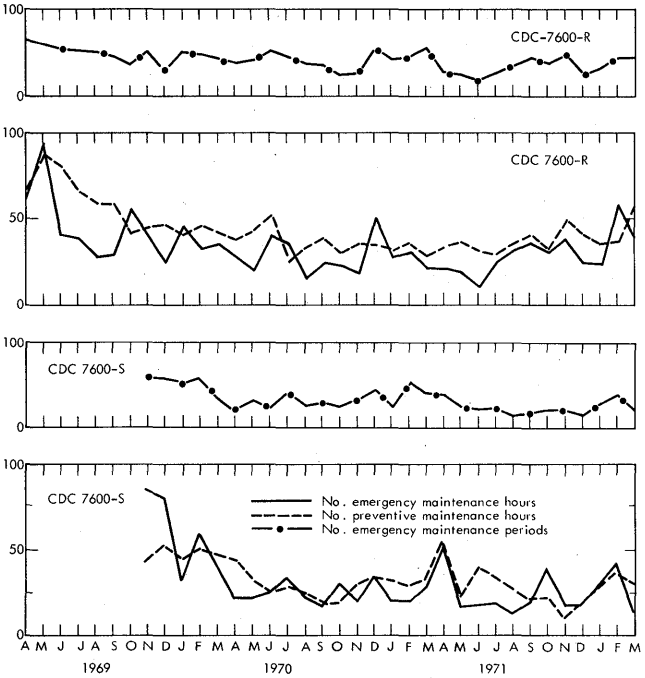


Figure 4—History of 7600 maintenance actions

The amount of PM time allowed is continually under review, and whenever the hardware shows an increased reliability, the PM periods are reduced.

Table 1 shows LLL's current preventive maintenance schedule and represents at best the current trade-off between the above maintenance philosophies.

Network availability during the prime usage periods

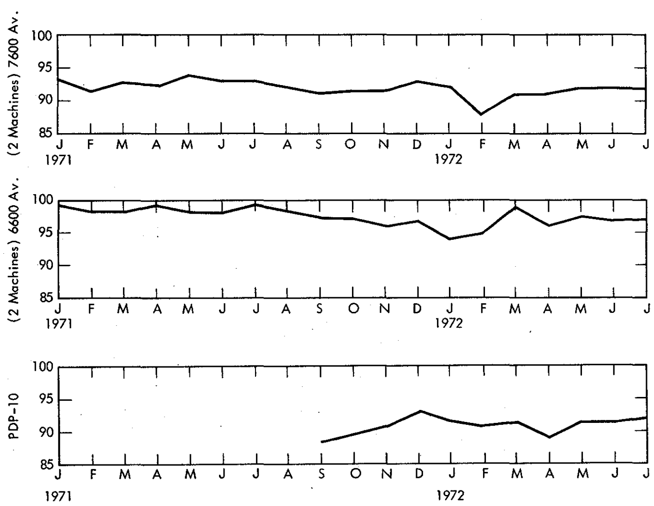


Figure 5—Percent of total machine availability

TABLE I—Current Scheduled Maintenance

Machine	Monday	Tuesday	Wednesday	Thursday	Friday	Interval
Network hub		x	x	x	x	4:00–8:00
CDC 6600	a					4:00–8:00
CDC 7600	x		x		b	4:30–8:00
CDC 7600	x		x		b	4:30–8:00
CDC 7600		x		x	b	4:30–8:00

[a] CDC 6600 taken on alternate Mondays.
[b] Any two CDC 7600's may be taken, but not all three at the same time.

is maximized by conducting scheduled maintenance at a time least visible to the user (0400-0800 hours) and by selecting subsets of components to be down concurrently.

For comparison, the scheduled or preventive maintenance (PM) and unscheduled or emergency maintenance (EM) history for the CDC 7600 R (serial No. 1) and CDC 7600 S (serial No. 2) host computers is illustrated in Figure 4. These maintenance actions required the host computers to be off-line and therefore completely unavailable to the user. Figure 5 shows the average total percentage availability for the CDC 7600 R and S host computers, the CDC 6600 L (serial No. 13) and CDC 7600 M (serial No. 31) host computers and the total percentage availability for the PDP-10 hub or control computer. The percentages are arrived as follows:

$$\frac{\text{Hours in Month—(PM+EM)}}{\text{Hours in Month}}$$

EVALUATION OF DIAGNOSTIC MAINTENANCE TOOLS AND PROCEDURES

The diagnostic maintenance tools do provide for rapid, positive identification and isolation when the component or device failure is solid. However, experience has indicated that most failures tend to be intermittent in nature and difficult to identify and isolate. Even though great amounts of time and money can be spent attempting to isolate intermittent failures, it has not been demonstrated at LLL that intermittent failures become significantly less intermittent when extensive off-line diagnostic periods are used. For this reason, it is concluded that it is better to catalog an intermittent error for administrative analysis, recover as softly as possible, and promptly return the device or component to full productivity rather than insist on the immediate off-line isolation of the problem.

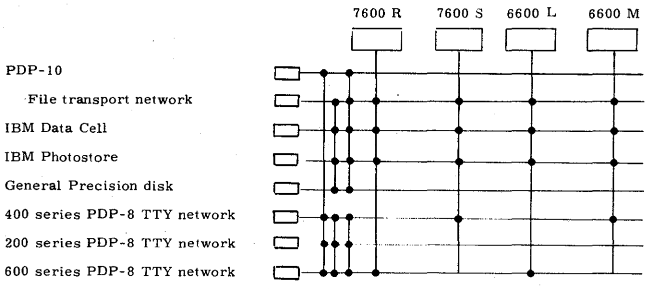


Figure 6—Host computers on-line during sampling period

Samplings (Figure 6) of system availability taken hourly Monday through Friday from 0800-1630 hours from November 1970 through April 1972 demonstrate the following:

	Percent
Total System Availability (all devices on line)	75*
Partial System Availability (Useful work being accomplished by at least one host)	100*

ACKNOWLEDGMENTS

The authors express their appreciation to LLL's Donald L. von Buskirk, Richard E. Potter, Robert G. Werner, and Pete Nickolas for their contributions to this paper.

REFERENCES

1 D L PEHRSON
An engineering view of the LRL Octopus computer network
Lawrence Livermore Laboratory Rept. UCID-51754 1970

2 *Livermore time-sharing system* manual M-026
Lawrence Livermore Laboratory 1972

3 K H PRYOR et al
Status of major hardware additions to Octopus
Lawrence Livermore Laboratory Rept UCID-30036 1972

* Power failures affecting the entire network are not included in these figures.

APPENDIX

Commands available include:

Printer/Punch

P ALL P1 Send all printer output to on-line printer No. 1. Printer 2 can now go down for maintenance.

P ALL P2 Send all printer output to on-line printer No. 2. Printer 1 can now go down for maintenance.

P ALL HSP Send all printer output to tape for off-line processing. Both on-line printers can now go down for maintenance.

P KILL P1 or P2 or PUNCH Abôrts processing of current printer/punch files on indicated device.

P2 HSP Send all printer No. 2 output to tape for off-line processing. Printer 2 can now go down for maintenance.

P NORMAL Restores operating status of printer output devices.

Disk

D N MMM N is a disk unit number 1 through 4. MMM is either "IN" or "OUT." This will allow or prevent, respectively, the creation of new files on disk N. If "OUT," existing files remain accessible and disk N can be scheduled for maintenance.

DP N MMM As described for the D option, but will also purge disk N of all existing files. All files on disk N are destroyed and are no longer accessible.

Drum

P DRUM DOWN Transfers system tables from the drum to memory and rewrites these tables to a disk file. All subsequent attempts to access the drum will be redirected to the disk. This provides backup capability for the drum and allows the drum to be taken down for maintenance.

P DRUM UP Restores normal operating status of the drum. System tables that have been stored on disk during the drum down period are transferred from disk to memory and rewritten to the drum.

Tape

C N Tape unit N is physically not available to the system.

F N Tape unit N is physically available to the system.

E P A tape error status is returned to program P.

X N Severs logical connection between tape unit N and problem program.

Execution

SP NNNNNN IN TEXT Allows only privileged user number NNNNNN access to the system. All users programs are saved on disk, and all users remote TTY stations are logged out. The TEXT is sent to all users attempting to log in.

SP NNNNNN OUT Removes privileged user number NNNNNN and automatically restarts previously running programs and restores normal time-sharing.

S TEXT Prohibits any additional log in. TEXT is sent to remote TTY stations.

R Restores normal time-sharing.

Broadcasts

I STORE TEXT Sends the TEXT to all logged in remote TTY stations. Sends the TEXT once to all remote TTY's at log-in time. Sends TEXT to TMDS.

I ERASE Erases the I STORE TEXT.

I BROAD TEXT Broadcasts TEXT once to all remote TTY stations and sends TEXT to TMDS.

The retryable processor

by GEORGE H. MAESTRI

Honeywell Information Systems Inc.
Phoenix, Arizona

INTRODUCTION

In the interest of improving readability, instruction retry is presented generically. Technical terms unique to the 6000 are avoided.

The intermittent failure problem

A hard failure is the result of a logic signal either remaining permanently in a one or a zero state or of a signal consistently switching to an improper state (such as an AND gate behaving as an OR). In the case of an intermittent failure, identical instructions executed in different sequences or at different times will not fail consistently.

Test and Diagnostic (T&D) programs are designed to diagnose solid failures and are successful at accomplishing their design objectives. They begin by certifying a basic core of processor functions and then read the T&D executive into memory to commence comprehensive testing. No function is used until it is tested. A problem with this approach is that intermittent failures can occur on functions that have been previously certified, completely destroying the rationale of the program.

The second, and most likely problem, is that T&D programs seldom detect intermittent failures. To trigger an intermittent failure, the T&D must execute a particular sequence of instructions in an exact order, using the proper data patterns. Also, intermittents are often triggered by stimulus that is beyond the control of programs; thermal variations, mechanical vibration and power fluctuations are examples. Sequence sensitive intermittents can be caused by the following: a low noise threshold in an IC, crosstalk, slow rise or fall times of logic signals, or extra slow or fast gates that activate a normally quiescent race condition.

A study performed by the U.S.A.F.[1] showed that 80 percent of the electronic failures in computers are intermittent. A second study performed by IBM[2] stated that "intermittents comprised over 90 percent of the field failures."

Alternatives for solution

The ideal method of diagnosing an intermittent failure is to bypass test programs and to diagnose directly from the symptoms of the original failure. The only reason that this method is not in common use is that the set of failure symptoms that are available to programs is inadequate for that purpose. First, it is necessary to know which bits are in error and whether they failed to switch from a one to a zero state or vice versa. Second, all control points should be visible to the diagnostic for all cycles of the failed instruction. A scratchpad memory or snapshot register could save the state of control points and data in case an error is detected.

In the case of intermittent failures, the problems associated with using a failing processor to diagnose its own ills will be minimal. Also, a minicomputer or a second processor could do the data processing necessary for diagnosis.

If it is not possible to diagnose from the symptoms of the original failure, it will be necessary to run a T&D program to gather additional information about the failure. However, T&D programs are ineffective against intermittent failures because they are usually unsuccessful in detecting them. What is required then is a method of making an intermittent failure solid.

Stress testing is often effective in changing an intermittent failure to a solid failure. Stress testing involves setting marginal voltage and timing conditions to amplify the effects of slow rise times, low switching thresholds and race conditions; thermal stress is also applied for the same reasons. Mechanical vibration is applied while the T&D is in execution to locate loose wirewraps, defective connectors, microphonic chips or substrates, conductive debris that is caught between wirewrap pins, and defective printed circuit runs.

Error Detection And Correction (EDAC) codes are particularly effective for correcting memory parity errors, which are inherently not recoverable by instruction retry techniques. Algorithms have been developed to allow single or multiple bit failures to be corrected. Some EDAC codes are particularly effective for correcting memory parity errors, which are inherently not recoverable by instruction retry techniques. Algorithms have been developed to allow single or multiple bit failures to be corrected. Some EDAC codes can traverse adders to correct addition errors.

Advantages of instruction retry over other alternatives

There is no reason that an immediate branch to a diagnostic program must exclude an instruction retry. The detection of an error can cause an immediate branch to a diagnostic program that will log and analyze all available symptoms. Failure analysis could result in the generation of a list of all logic elements whose failure could result in the symptoms that were recorded. The boundary between suspect and nonsuspect logic will be called "limits."

Once limits are established, they can be analyzed to determine if the failure has been sufficiently isolated to enable a repair. If they have not, the processor can be restored to the state that existed prior to the failure and the instruction can be retried. If the retry attempt is successful, the processor will remain available to the customer until the next failure. Subsequent failures will serve to further narrow the limits by contributing new symptoms.

Stress testing requires that the processor be dedicated to T&D, which means that the processor will not be available to the user. Instruction retry will keep the processor available to the user as long as it is successful; maintenance can be performed during slack time. Also, thermal and mechanical stress can inflict new damage.

While EDAC is an effective means of correcting memory parity errors, it is incapable of correcting control point errors in the processor. If a word of data and its correction code fail to traverse a switch, because of a control point error, both the correction code and the data will be lost.

Since instruction retry is conversely ineffective against memory parity errors and most effective against control point errors, EDAC and instruction retry will complement each other.

Obstacles to retry

A prerequisite to a successful instruction retry is that memory locations and registers associated with the faulted instruction must contain the same data that they did before the instruction was started. If a register or memory location was changed during the execution of the instruction, it must be restored before retry can be attempted. Restoration will not be possible if the contents of a memory location is added to and replaces the contents of a register and an image of the original register contents is not available.

Memory parity errors are detected after an error has invalidated the contents of a memory location. Unless memories are duplicated or EDAC is present, memory parity errors cannot be retried.

The instruction repertoire of some processors includes instructions that cause the memory controller to perform a destructive read of a memory location. If an error occurs on an instruction that caused a destructive read, it will be necessary to restore the cleared memory location before retry can be attempted.

A MOVE is an instruction that moves a block of data from one area of memory to another. If a MOVE overlays part of its source data, instruction retry will not be possible. For example: if 100 words are moved from location 70 to location 0, words 70 to 100 of the source data can be overlaid.

Indirect addressing offers the programmer the capability to address operands via a string of indirect words that are often automatically updated every time they are accessed. If a faulted instruction has obtained its operand via such a string, every indirect word in the string must be restored prior to retrying the instruction. If an error occurred during an indirect word cycle, then only the indirect words preceding the failure must be restored.

In processors with hard-wired control logic, the multicycle instructions repeatedly change the contents of registers as fast as data can be cycled through the adder. Delaying every cycle for error checking is often an unacceptable degradation of throughput. Consequently, a register could be overlaid with erroneous data before the error can be detected.

Instruction overlap is a feature of large scale processors that complicates identifying the failing instruction. Instruction overlap takes advantage of the fact that no single instruction uses all of the processor logic at any given instant. While one instruction is terminating, a second instruction may be using the adder, while a third is undergoing an address preparation cycle and a fourth is being fetched from memory. If instruction overlap is active, the instruction counter may be pointing to the instruction being fetched from memory at the time an error is detected on the instruction that is in the address preparation sequence. Thus, merely safestoring the instruction counter at the time of failure will not serve to identify the failing instruction.

Design methodology to avoid obstacles

The destruction of data can be avoided for single cycle instructions by not overlaying the register in the first place. The adder sum can be buffered or merely retained on the data lines until error checking has finished. If an error is detected, the instruction can be aborted before the defective data is moved into a register.

EDAC can enable the recovery of memory parity errors.

At the time of a fault, the state of the cycle control flags and address register could be saved in a snapshot register. The contents of the snapshot register could identify the failing cycle of a MOVE so that software could continue moving the block of data in place of the interrupted MOVE. This will be effective in recovering an error on a MOVE that has overlaid part of its source data. The snapshot register can also serve as a diagnostic aid by saving the state of cycle control flags at the time of an error.

One method of restoring a string of indirect words is to obtain a pointer to the first indirect word from the address field of the instruction. Since indirect word updates are performed by a fixed and known algorithm, it will be a simple matter to restore the first word of the string to obtain a pointer to the second and then follow the string; restoring each word to obtain a pointer to the next.

However, several pitfalls exist in the above method. One is that if the error occurred on an indirect word cycle, the recovery program must know when to terminate its indirect word restoration activity. Otherwise, it may restore indirect words that have not been updated, thus inducing an error. Also, the recovery program must be able to determine if the indirect word being restored has been damaged by a parity error. Another problem is the possibility that an indirect string may wrap back on itself, causing a word to be updated twice. If the recovery program merely follows the string, without knowledge of the double update, it will fail to restore part of the string and will induce an error when the instruction is retried.

An alternative that would also allow software to restore indirect words without inducing errors, is to provide a scratchpad memory to save the state of sequence control flags and memory addresses for every cycle of an instruction. If an indirect word string wraps back on itself, causing an indirect word to be updated twice, it would present no problem to the recovery program; the snapshot register will contain two entires for that word, and it will be rolled back twice.

Providing intermediate registers will serve to both increase processor speed and to protect the contents of primary registers in case of error. The secondary registers can be placed at the inputs to the adder to hold the operand from memory and the operand from a primary register. The secondary registers will also serve to decrease the execution time of multicycle instructions, by providing a shorter path to the adder. Intermediate registers will allow date manipulation to be performed for multiplies, divides, etc., without changing the contents of the primary registers. The sum, product, quotient, etc., can be moved into a primary register after error checking is complete.

Another alternative would be to save an image of the registers every time that an instruction comes to a normal (error-free) termination. Instruction retry could be accomplished by refreshing the primary registers with data from the backup registers.

If the processor has instruction overlap capability, it will be necessary to correct the instruction counter when a fault is detected. Otherwise, it may not be possible to determine which of several instructions, that are simultaneously in execution, failed.

Another possibility is to provide an instruction counter for each of the instructions that can be simultaneously executed. The instruction counter assigned to the failing cycle can be selectively safestored.

A third possibility is to include a failure flag in the scratchpad memory to identify the failing cycle. If the failing cycle is identified in the scratchpad, the instruction containing that cycle can be identified by a program.

Tradeoffs

Figure 1 shows that the simple processor operations; i.e., loads, stores, transfers and instruction fetches account for 95 percent of all processor operations (excluding address modification). Figure 2 shows that 30 percent of all processor operations are preceded by some type of address modification; of the 30 percent, 25.4 percent is simple register type modification and 4.6 percent involves indirect words. Since register type address modification does not in itself alter register contents, it is not a factor in determining the retryability of a simple instruction. Consequently, if instruction retry is implemented at all, it will be better than 90 percent effective.

The mandatory design requirements for instruction retry are:

(1) The failure must have been detected by hardware error detection.
(2) The failing instruction must be identifiable.
(3) Instruction operands must either remain intact or must be restorable.

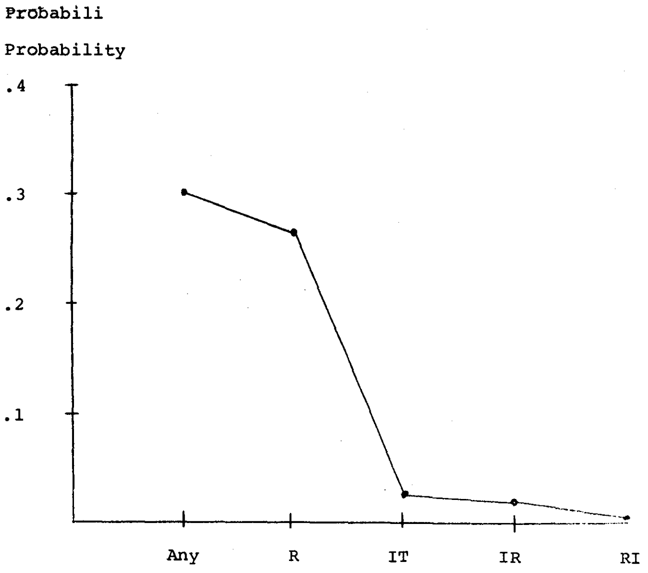
The simple mechanism of holding the adder sum output on the data lines until it has been determined that an error has not occurred will prove effective against processor/memory interface errors, for simple instructions. If the processor does not have instruction overlap capability, merely safestoring the instruction address in a predetermined memory location will serve to identify the failing instruction.

If the processor has overlap capability, then a more sophisticated method of identifying the failing instruction is necessary. (See "Design Methodology To Avoid Obstacles"). Once the failing instruction is identified, the opcode and address modification specifier can be examined to determine if the instruction is a candidate for an instruction retry attempt.

Number of Operations	2,653,856
Number of Instructions	1,661,723
Instruction Fetches**	938,873
Stores	367,811
Multiplies	1,196
Divides	933*
Transfers	479,421
Shifts	41,894
Floating Adds	3,231*
Floating Multiplies	2,621*
Floating Divides	372*
Loads	743,170
Load Register, Store Register	1,078
Negate	450
Master Mode Entry	661
Execute Double, Execute	4,152
Repeats	5,326
Repeated Instructions***	53,260*
Returns	4,088
Binary Coded Decimal	2,919
NOP	2,400
Retryable Operations	2,589,287 (97.5 percent)

* Not Retryable 64,569 (2.5 percent)
** Instruction Fetches = 56.5 percent Number of Instructions
*** Repeated Instructions = Repeats times 10

Figure 1—Instruction frequency analysis



	Totals
Any address modification	761,580
Address modification = R	644,062
Address modification = IT	58,161
Address modification = IR	53,778
Address modification = RI	5,579

Figure 2—Probability of address modification on processor operations[3]

The addition of a snapshot register bank and other features mentioned in the "Design Methodology" section of this paper will allow multi-cycle instructions and instructions utilizing indirect address modification to be retried. This will raise the effectiveness of instruction retry to almost 100 percent.*

Cost of implementation

The 6000 processor features hard wired control logic and instruction overlap. Its four instruction counters, scratchpad register bank and intermediate registers allow instruction retry to be better than 97 percent successful (see Figure 1).

* 100 percent effectiveness means that instruction retry can be unconditionally attempted.

However, none of the above features were incorporated as instruction retry aids and cannot be considered a cost of implementation. The instruction retry effort was not started until after the processor design was frozen.

The four instruction counters are an improvement on the 600 line's "ICT Correction" logic. It has always been considered good design practice to accurately identify the location of a fault. The scratchpad register bank was implemented in approximately 400 SSI chips as a dump analysis and T&D aid. The intermediate registers were implemented to speed processor instruction execution.

The instruction retry feasibility study, programming and debug efforts required one man year.

CONCLUSION

If a processor failure is detected, there are two possible actions that can be taken. One is to abort the program that was in the execution, to prevent propagation of the error. The second is to retry the failing instruction in an attempt to bypass a possible intermittent failure. Since 80 to 90 percent of processor failures are intermittent, there is an excellent chance that instruction retry will succeed.

The advantage of retrying the instruction over aborting the program is that a successful instruction retry will keep the system available to the customer while an abort takes it away from him.

As long as errors continue to be successfully recovered and performance is not seriously degraded, maintenance can be deferred until a convenient time period. The question of what comprises a serious degradation is probably best answered by the individual user. In a real time application, 3 or 4 percent may be serious; while in an I/O bound batch application, 30 or 40 percent degradation may be tolerable.

REFERENCES

1 J P ROTH
Phase II of an architectural study for a self-repairing computer
USAF Space and Missile Sys Org Los Angeles
CA AD 825 460 18 1967

2 M BALL F HARDIE
Effects and detection of intermittent failures in digital systems
IBM No 67-825-2137 1967

3 K ROSENSTEEL
An analysis of dynamic program behavior
Honeywell No ASEE #54 1972

ACKNOWLEDGMENTS

Peter J. Scola

For originating many of the hardware changes that enabled instruction retry to be implemented on the Honeywell 6000 line and for obtaining the necessary funding.

Harlow E. Frick

For assistance in implementing the Honeywell Instruction Retry Program.

APPENDIX

With the exception of the footnotes in Figure 1, Figures 1 and 2 were extracted from a report by Kenneth Rosensteel[3] entitled: "An Analysis of Dynamic Program Behavior". Figures 1 and 2 represent the total number of instructions executed by a mix of FORTRAN, ALGOL and COBOL compilations and executions.

In considering address modification, Figure 2 shows the four possible modification types:

> *Register* (R)—Indexing according to the named register and termination of the address modification procedure.
> *Register Then Indirect* (RI)—Indexing with the named register, then substitution and continuation of the modification procedure as directed by the tag field of this indirect word. (Indirection with pre-indexing.)
> *Indirect Then Register* (IR)—Saving the register designator, then substitution and continuation of the modification procedure as directed by the tag field of this indirect word. (Indirection with post-indexing.)
> *Indirect Then Tally* (IT)—Indirection, then use of the indirect word as tally and address with automatic incrementing and decrementing.
> *Any*—Probability of any type of address modification.

Evaluation nets for computer system performance analysis

by G. J. NUTT*

University of Washington
Seattle, Washington

INTRODUCTION

The growing complexity of modern computer systems has made performance evaluation results more and more difficult to obtain. The difficulty of representation and analysis of combination hardware/software systems has increased with the level of sophistication used in their design. One popular approach that has been used for evaluating proposed computer systems is simulation.[1]

In this paper, a method of representation is presented that is useful in constructing a modular model, where the level of detail may vary throughout. This method has been designed to aid implementation of these models, with a net effect of providing the ability to construct flexible simulation models in a relatively small amount of time. A graphic approach is used so that the two dimensional structure of the machine is pictorially available to the simulation designer. The graphs are also useful for planning measurements of either a simulation model or the machine they represent.

An evaluation net is made up of *transitions* interconnected by directed edges with *locations*. Each location may contain a *token*. For a particular transition, the members of the set of locations directed into the transition are called *input locations*, and the members of the set of locations directed away from the transition are called *output locations*. A transition *fires* if the set of input and output locations satisfies the definition of that particular transition, causing one token to be removed from each location of a prespecified subset of the input locations and one token to be placed on each location of a prespecified subset of the output locations.

Example

Figure 1(a) shows an example with two transitions. The first transition, (the vertical line labeled a_1), has two input locations, (the circles labeled b_1 and b_2). The second transition, a_2, has a single input location, b_3, and two output locations, b_1 and b_4. For this example, let a dot in a location represent a token residing on that location. Suppose that the definitions of the two transitions, a_1 and a_2, specify that they fire when all input locations contain a token and all output locations do not contain a token. Then in Figure 1(a), transition a_1 is ready to fire. Figure 1(b) shows the same transitions and locations after transition a_1 has fired. Figure 1(c) shows the result after firing a_2. In this example, the prespecified subsets are the complete sets of input and output locations.

Figure 1 may be interpreted as the following situation in a computer system. If b_1 contains a token, the central processor is available. If b_2 contains a token, there is a job requesting the central processor. Thus, concurrent occupancy of tokens on b_1 and b_2 indicates that there is a request for the central processor and it is available, causing transition a_1 to take action, (representing central processor allocation). The time required to fire a_1 indicates allocation time, and is negligible. A token on b_3 represents central processor activity. The transition time for a_2 reflects the length of central processor time for the job, and on completion of firing, the central processor again becomes available, (i.e., a token is placed on b_1) and the job has completed central processor utilization, (i.e., a token is placed on b_4).

Evaluation nets are derived from the work of Petri[2] and Noe[3]; they have also been influenced by the work of Holt,[4,5] who is primarily responsible for the development of Petri nets. The nets given in this paper differ from Petri nets in their' 'practicality." The path of a token through a net is well-defined by providing a mechanism to choose from a set of alternate paths that the token might take. Each token in the net is distinct and retains its identity. The token may have a vector of attributes, (capable of taking on values), that may be modified by the various transitions that operate on the token. The time required for each execution of a transition is part of the specification of the net, thus introducing time as a measure of net performance.

* Present address: Department of Computer Science, University of Colorado, Boulder, Colorado 80302

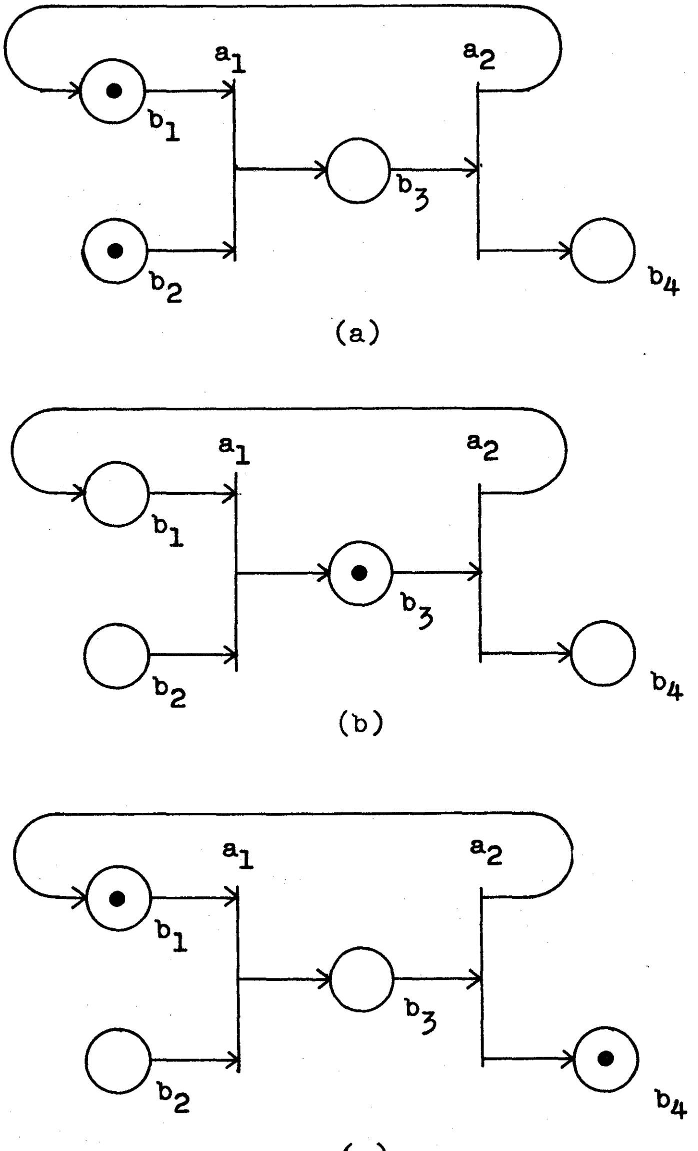


Figure 1—Transition firing

The primary predecessor of evaluation nets,[3] has provided the motivation for this development. These predecessor nets, called modified Petri nets, were used to describe a simulation model of a CDC 6400 in a validation study.[6] The modified Petri nets have shortcomings in the area of describing action on a specific token, (representing a job in a computer system). These deficiencies manifest themselves in indeterminate path flow of the token and the lack of quantitative properties in the network. Evaluation nets are not only useful in describing a simulation model, but also in constructing the model. The additions made to the previous work allow direct interpretation of the description to yield a working simulation model.

THE CLASS OF EVALUATION NETS

In this section we shall describe the class of evaluation nets in detail. We begin by defining location types and statuses. All locations are connected to at least one transition. If a location is an input (output) location for some transition and is not an output (input) for any other transition, the location is said to be *peripheral*, e.g., b_2 and b_4 in Figure 1. If a location is not peripheral, it is an *inner location.* A location is *empty* if it does not contain a token, and *full* if it contains a token, e.g., locations b_1 and b_2 are full and b_3 is empty in Figure 1(a). If it is not known whether the location is empty or full, the status of the location is *undefined.* An inner location may change from empty to full or full to empty only by the firing of one of the transitions to which it is connected. The conditions for the status of a location to be undefined will be discussed later, as will the utility of this convention.

Transition schemata

A transition definition is given by a triple, $a=(s, t(a), q)$, where s is a *transition schema* (or *type*), $t(a)$ is a *transition time,* and q is a *transition procedure.* The movement of tokens from input locations to output locations is described by the schema. The number of output locations is limited to two, the number of input locations is limited to three, and the number of connected locations is limited to four for any given transition. If a location is empty, its status is denoted as "0". If the location is full, the status is denoted as "1". The undefined status is given by the symbol, "Φ". The status of the set of locations associated with a transition is given as an ordered p-tuple of individual location statuses, where $2 \leq p \leq 4$. The action of the transition is exhibited by mapping the set of statuses before any action takes place into the set of statuses after action takes place. If a particular p-tuple is mapped into itself, the transition effectively takes no action for that status configuration. Otherwise, the map defines token removal and placement during transition firing. In the schema definition, p-tuples mapped into themselves are omitted for brevity. For example, a transition modeling an "AND gate with reaction" is described by the map

$$J(a, b, c): \quad (1, 1, 0) \rightarrow (0, 0, 1)$$

where the locations a and b are input locations and location c is an output location. The interpretation of the mapping, J, is that if location a and location b are full and location c is empty, a metamorphisis of the

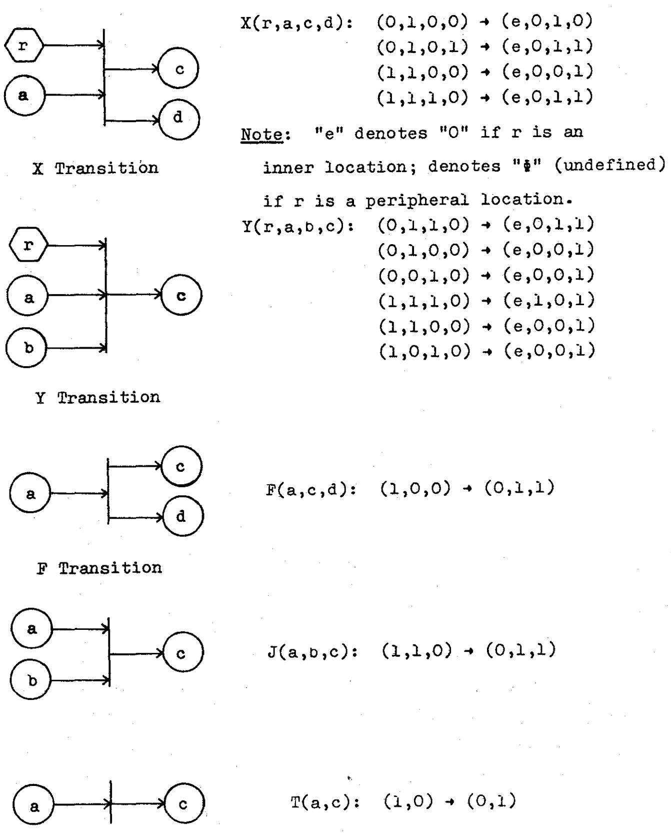


Figure 2—Transition schema graphs

location statuses takes place, resulting in locations a and b becoming empty and location c becoming full. No other p-tuple of location statuses causes an action by the transition. In Figure 1, transition a_1 corresponds to this example.

Let $\{r, a, b\}$ be the set of input locations and $\{c, d\}$ be the set of output locations for the following definition (see Figure 2 for graphical representation). Define the following schema:

$$\begin{array}{rl}
X(r, a, c, d): & (0, 1, 0, 0) \rightarrow (e, 0, 1, 0) \\
& (0, 1, 0, 1) \rightarrow (e, 0, 1, 1) \\
& (1, 1, 0, 0) \rightarrow (e, 0, 0, 1) \\
& (1, 1, 1, 0) \rightarrow (e, 0, 1, 1) \\
Y(r, a, b, c): & (0, 1, 1, 0) \rightarrow (e, 0, 1, 1) \\
& (0, 1, 0, 0) \rightarrow (e, 0, 0, 1) \\
& (0, 0, 1, 0) \rightarrow (e, 0, 0, 1) \\
& (1, 1, 1, 0) \rightarrow (e, 1, 0, 1) \\
& (1, 1, 0, 0) \rightarrow (e, 0, 0, 1) \\
& (1, 0, 1, 0) \rightarrow (e, 0, 0, 1) \\
F(a, c, d): & (1, 0, 0) \rightarrow (0, 1, 1) \\
J(a, b, c): & (1, 1, 0) \rightarrow (0, 0, 1) \\
T(a, c): & (1, 0) \rightarrow (0, 1)
\end{array}$$

The symbol "e" in the range of the map, (the right hand side of the transition schema definitions), is "0" if r is an inner location and "Φ" otherwise. Let us characterize the schema as graphs, where a circle represents a location, a vertical line represents a transition, and a hexagon represents a *resolution location*, (see Figure 2). A resolution location, r, is an input location, (the first coordinate) for a transition of type X or type Y. Consider the description of the type X transition: r is a resolution location and a.is a "normal" input location. If the status of r is empty, a is full, and the output location, c, is empty, the transition fires, removing a token from location a and placing a token on location c, (labeled by a zero in the graph of Figure 2). The status of location d is essentially ignored for the case of r being empty. If the resolution location is an inner location, the transition firing leaves the status empty. *Otherwise the status is left undefined.* When the status of r is full, the transition places a token on location d, (labeled by a one in the graph), regardless of the status of location c. Hence, the resolution location acts as a switch, routing tokens to one of two alternate output locations, c or d. If it is an inner location, its switching action is controlled by internal operation of the net. If it is a peripheral location, switching is controlled by a resolution procedure which allows influence from tokens, as will be discussed later.

The type Y transition uses the resolution location, r, in a slightly different way. Note that locations r, a, and b are input locations. If location a is full and locations b and c are empty, a token will be removed from a and placed on c for r being either 0 or 1. Similarly, if location b is full and locations a and c are empty, location b yields a token to location c for r being either empty or full. The setting of r is critical only if locations a and b each contain a token simultaneously. If r is set to 1, b yields the token to c. If r is set to 0, a yields the token to c.

The F, J, and T transitions operate whenever all input locations are full and all output locations are empty. Transition firing causes the removal of a token from each input location and placement of a token on each output location. Figure 1 illustrates the F and the J transitions.

Each transition in an evaluation net must satisfy one of the schemata given above. If it is necessary to model a procedure with more than two inputs (or outputs), transitions may be combined to produce the effect of multiple inputs (or outputs). For example, Figure 3 represents a procedure, (or event) for which all three inputs must contain tokens to complete the firing of the transition labeled a_2.

The status of a location may be given symbolically

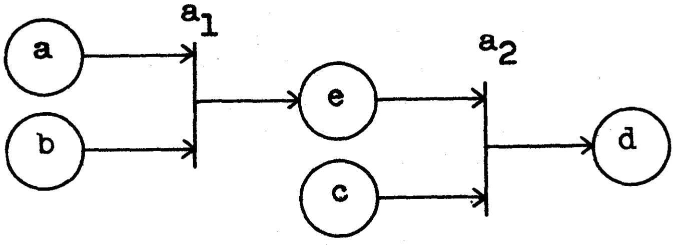


Figure 3—Multiple input processes

by providing a mapping, M, of the set of locations into the set of statuses, $\{0, 1, \Phi\}$. For example, if b is a location, $M(b)=0$ if b is empty; $M(b)=1$ if b is full; $M(b)=\Phi$ if the status of b is undefined.

In the transition schema definition it was shown how a peripheral resolution location becomes undefined. Since no transition can fire when one of its associated locations is undefined, a *resolution procedure* is required to set the status of a peripheral resolution location to either empty or full. The resolution procedure, thus, is a mechanism for communication between the environment and the net. A resolution procedure is an expression of the form

$$r\colon\ [p_1 \rightarrow M(r):=i;\, p_2 \rightarrow M(r):=1-i]$$

where i is either 0 or 1, r is the label of the peripheral resolution location, and p_1, p_2 are "Algolic" Boolean expressions (predicates) which can be evaluated to either true or false, (Nutt[7] contains a more complete handling of these predicates). The resolution procedure is evaluated by first evaluating p_1. If it is true, $M(r)$ becomes i and further evaluation of the procedure is discontinued. Otherwise, p_2 is evaluated; if it is true, $M(r)$ is set to $1-i$. In either case the resolution procedure evaluation is discontinued after predicate p_2 is evaluated. Note that when both predicates are evaluated as false, the marking of r remains undefined. The procedure need not be evaluated again until one of the arguments of the predicates changes its status. Examples of resolution procedures are given in the next section.

Token attributes and their modification

The transition schema definitions imply that no location may contain more than a single token at a time, provided that an initial marking does not place more than one token on any location. For example, the type T transition fires only when the input location is full and the output is empty, hence only an empty location can receive a token. This property of evaluation nets, (known as safety), allows each token to be distinct. Since tokens retain their identity, we shall give them names and associate a list of n attributes with each token, such that each attribute may take on a value. A token, K, with n attributes is denoted as $K[n]$, and if location b contains $K[n]$, we shall write $M(b)=K[n]$ rather than $M(b)=1$. The jth attribute of the token K is denoted as $K(j)$. At times we will find use for tokens with no attributes, whose identity is unimportant. We shall continue to indicate these tokens by the symbol "1". For example, a resolution location setting will only need to indicate empty or full status, hence can be denoted as $M(r)=0$ or $M(r)=1$.

The attributes of a token impose a data structure on the locations of a net. Any particular location will always receive (and yield) tokens with a fixed number, n, of attributes. A location, b, which holds tokens with n attributes is denoted as $b[n]$. Hence, more properly, the expression of a marking should be $M(b[n])=K[n]$. As long as the context makes the dimension of b clear, we will not insist on the more complete notation. Conceptually, a location $b[n]$ is composed of n "slots" which contain the n attributes of a token residing on the location. The values of the slots are the values of the corresponding attributes. If the location is empty, the values of the slots are undefined. We shall refer to the ith slot as $M(b(i))$, hence if $M(b[n])=K[n]$, the ith attribute of K may be denoted $M(b(i))$.

Let $b[m]$ be an output location of transition a_i and an input location of transition a_j (see Figure 4). First, suppose that a_i produces a token, $K[n]$, to be placed on $b[m]$, where $n \neq m$; the resulting $M(b[m])$ is defined as follows. Let g be the minimum of the integers n and m. Then

$$M(b(1)):=K(1)$$
$$M(b(2)):=K(2)$$
$$\vdots$$
$$M(b(g)):=K(g)$$

If n is greater than m, then the remaining attributes of $K[n]$ are lost. If n is less than m, then the values of $M(b(i))$, for $n+1 \leq i \leq m$, are undefined. Next suppose that a_j removes the token on $b[m]$. The number of

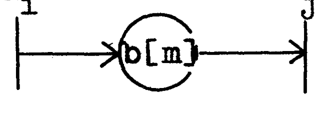


Figure 4—Number of attributes

attributes for that token is defined to be m; where $M(b(n+1)), \ldots, M(b(m))$ are undefined through the placement of the token on $M(b[n])$.

Although the transition schema of a particular transition defines the locations that are to receive and yield tokens, the identities and attribute modifications are not reflected without specifically providing for them. For example, suppose a transition $a=(s, t(a), q)$, has a schema, s, of $J(b_1[n], b_2[n], b_3[n])$ and $M(b_1)=K_1[n]$, $M(b_2)=K_2[n]$, where $K_1[n]\neq K_2[n]$. A transition procedure has the form

$$[p_1 \rightarrow (e_{11}; e_{12}; \ldots ; e_{1n}) : \ldots : p_k \rightarrow (e_{k1}; e_{k2}; \ldots ; e_{km})]$$

where the p_i are predicates $(1\leq i\leq k,\ k$ finite), as described previously, and the e_{ij} are "Algolic" arithmetic assignment statements, e.g.,

$$M(b_3(4)) := M(b_1(4)) + 100.$$

A transition procedure is evaluated by the following algorithm:

1. Set i to 1. Go to step 2.
2. If p_i is true, execute $(e_{i1}; e_{i2}; \ldots ; e_{ij})$ and then terminate transition procedure evaluation. Otherwise go to step 3.
3. Set i to $i+1$. If i is greater than k, terminate the transition procedure evaluation. Otherwise go to step 2.

Transition firing

A transition firing may now be more formally defined as consisting of the following phases:

pseudo enabled phase: A transition is pseudo enabled if all locations satisfy the left hand side of a schema except for the undefined status of a peripheral resolution location. Since this status is undefined, the resolution procedure must be evaluated. (The resolution procedure cannot be evaluated unless the transition is pseudo enabled.)

enabled phase: A transition is enabled if all location statuses satisfy the left hand side of a schema. The transition then begins operation.

active phase: Transition action is in progress. The status of all associated locations does not change.

terminate phase: The transition completes processing, changing the status of output locations to agree with the right hand side of the schema, then executing the transition procedure, and finally changing the status of the input locations to agree with the right hand side of the schema.

The existence of an active phase in a transition firing implies an associated time that the transition requires to carry out its operation. An expression reflecting this time is provided by the second coordinate of the transition description, $(s, t(a), q)$. This specification, $t(a)$, may be a constant value, or it may be a function that is evaluated, (on entering the active phase), for the particular token(s) that enable the transition for a specific firing. It is convenient to express $t(a)$ for a transition of type X or Y as an ordered pair, where the first coordinate is $t(a)$ if the token moves from the location labeled "0" in a Y transition graph and the second coordinate is $t(a)$ if the token moves from the location labeled "1". In the X transitions, the first coordinate indicates $t(a)$ if the token moves to the location labeled by a "0" in the graph and the second coordinate indicates $t(a)$ if the token moves to the location labeled with a "1".

Since tokens that enable a transition reside on the input location(s) during the active phase, transition time imposes a *dwell time* on each location. The dwell time of a location, b, denoted $d(b)$, is the total amount of time any token resided on location b. The dwell time contributed by a particular token may be greater than the corresponding transition time for the token, since the token may have begun residence on the location without enabling the associated transition. The accumulation of dwell time for a location reflects the "occupancy time" or "busy time" for that location. Dwell times for a particular token may be summed up to provide a measure of the time required for that particular token to traverse the network, hence turnaround time. In Nutt,[7] measures of dwell time and their relationship to transition times are explored further.

Definition of an evaluation net

With the above preliminaries in mind, we can now define an evaluation net. An *evaluation net* is a connected set of locations over the set of transition schema and is denoted as the 4-tuple

E = (L, P, R, A) and an initial marking of the locations, M, where
L = A finite, non-empty set of locations.
P = A set of peripheral locations, $P\subseteq L$.
R = A set of resolution locations, $R\subseteq L$.
A = A finite, non-empty set of transition declarations, $\{a_i\}$,
a_i = $(s, t(a_i), q)$ where s is a transition schema, $t(a_i)$ is a transition time, and q is a transition procedure.

EXAMPLE OF AN EVALUATION NET

Let us construct a model of a very simple computer system which uses most of the concepts presented in the previous section. In our computer system, a job entering the mix may either wait for a single tape drive if it requests one, or if no tape drive is needed, proceed directly to processing by requesting the central processor. When processing is complete, the job relinquishes the central processor and releases the tape drive if it has been allocated.

In the description given below, we shall use the symbol "T" to denote a predicate that is always true. For the transition procedures that are implied by the transition schema, (i.e., there is no attribute modification and the transition merely copies the token from an input location to the output location(s) indicated by the schema), the procedure is indicated by a hyphen, "-".

Tokens that represent jobs in the computer system will be of the form $K[3]$, where

$K(1)$ = The number of tape drives required, (0 or 1).
$K(2)$ = Time required to fetch and mount a tape.
$K(3)$ = Central processor time.

Let $E=(L, P, R, A)$ be the net, (see Figure 5)

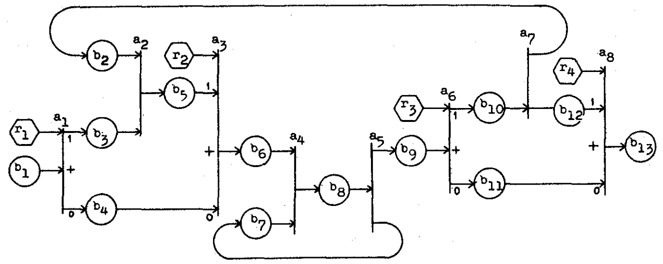
$$R = \{r_1, r_2, r_3, r_4\}$$
$$P = \{b_1[3], b_{13}[3]\} \cup R$$
$$L = \{b_2, b_3[3], b_4[3], b_5[3], b_6[3], b_7, b_8[3], b_9[3], b_{10}[3], b_{11}[3], b_{12}[3]\} \cup P$$
$$A = \{a_1, a_2, \ldots, a_8\}$$
$$a_1 = (X(r_1, b_1[3], b_4[3], b_3[3]), (0, 0), -)$$

i.e., a_1 is a type X transition with input locations r_1 and $b_1[3]$, which copies tokens to either $b_3[3]$ or $b_4[3]$ with no time delay.

$$a_2 = (J(b_2, b_3[3], b_5[3]), M(b_3(2)), [T \rightarrow (M(b_5[3]) := M(b_3[3]))])$$

i.e., a_2 is a type J transition whose time is determined by the second attribute of the token on the input location, $b_3[3]$.

$$a_3 = (Y(r_2, b_4[3], b_5[3], b_6[3]), (0, 0), -)$$
$$a_4 = (J(b_6[3], b_7, b_8[3]), 0, [T \rightarrow (M(b_8[3]) := M(b_6[3]))])$$
$$a_5 = (F(b_8[3], b_9[3], b_7), M(b_8(3)), [T \rightarrow M(b_7) := 1])$$
$$a_6 = (X(r_3, b_9[3], b_{11}[3], b_{10}[3]), (0, 10 \text{ sec.}), -)$$



b_1: Job ready to enter mix
b_2: Tape drive is available
b_3: Job requires tape drive
b_4: Job does not require tape drive
b_5: Tape job has drive allocated
b_6: Job requesting CP
b_7: CP is idle
b_8: CP is busy
b_9: Job is through with CP

b_{10}: Tape job ready to release drive
b_{11}: Non-tape job ready to vacate
b_{12}: Tape job ready to vacate
b_{13}: Job is complete
r_1: Routes tape job to b_3; Non-tape to b_4
r_2: Chooses job from b_5 or b_4
r_3: Routes tape job to b_{10}; Non-tape to b_{11}
r_4: Chooses job from b_{12} or b_{11}

Figure 5—Graph of evaluation net

$$a_7 = (F(b_{10}[3], b_2, b_{12}[3]), 0, [T \rightarrow (M(b_2) := 1)])$$
$$a_8 = (Y(r_4, b_{11}[3], b_{12}[3], b_{13}[3]), (0, 0), -)$$
$$r_1: [(M(b_1(1)) = 1) \rightarrow M(r_1) := 1; \; (M(b_1(1)) = 0) \rightarrow M(r_1) := 0]$$

i.e., r_1 takes on the same values as the first attribute of the token on $b_1[3]$.

$$r_2: [T \rightarrow M(r_2) := 1]$$

i.e., r_2 is always marked with a one.

$$r_3: [(M(b_9(1)) = 0) \rightarrow M(r_3) := 0; \; T \rightarrow M(r_3) := 1]$$
$$r_4: [T \rightarrow M(r_4) := 1]$$

Initially, let $M(b_2) = M(b_7) = 1$.

A job enters the net at location $b_1[3]$, (the arrival rate of subsequent jobs is not specified in this example). The existence of a token on $b_1[3]$ pseudo enables transition a_1 since $b_3[3]$ and $b_4[3]$ are both empty. Resolution procedure r_1 is evaluated, its marking being determined by the first attribute of the token on $b_1[3]$. Suppose that $M(b_1(1)) = 1$. Then the token is moved to location $b_3[3]$, the transition time being negligible, i.e., $t(a_1)$ is zero. Since $M(b_2) = 1$ initially, transition a_2 is enabled and becomes active. The transition time for a_2 is provided by the second attribute of the token on location $b_3[3]$ (which, let us say, contains "trace data" giving the time required to mount a tape). When this transition time has elapsed, $b_5[3]$ receives the token from $b_3[3]$, (see the transition procedure for a_2). The resolution location, r_2, is a "tie breaker" and in this case always favors jobs that have just had the tape drive allocated to them, should two jobs be ready to start requesting the central processor simultaneously.

The remainder of the net may be interpreted in the manner described above.

Let us suppose that the net was put into operation at time t_0 and was halted at time t_n. The elapsed time, $t_n - t_0$, is called the *system up time* and is denoted T_u. Notice that the dwell time of location b_7, $d(b_7)$, gives the central processor idle time and corresponds to $T_u - d(b_8[3])$. Similarly, the resource utilization of the tape drive is available from $T_u - d(b_2)$. If token $K_m[3]$ enters location $b_1[3]$ at time t_{im} and enters location $b_{13}[3]$ at time t_{jm}, the expression $t_{jm} - t_{im}$ reflects the turnaround time of the job represented as $K_m[3]$. Let $K_1[3]$, $K_2[3]$, . . . , $K_N[3]$ be N tokens that traversed the net. Then the mean turnaround time for this mix is given by

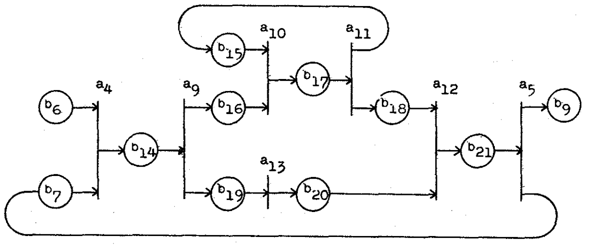
$$\sum_{m=1}^{N} (t_{jm} - t_{im})/N$$

or, alternatively, may be computed by summing up the appropriate dwell times and dividing by N.

The throughput rate may be expressed as

$$N/T_u \text{ jobs/system up time}$$

Suppose we exercise our model and find that it is insufficient for our purposes, e.g., disk access is completely ignored, but has an affect on the parameters we are measuring. We may choose to change the level of detail of the central processor activity in the net. Figure 6 suggests a slightly more complex net that reflects simultaneous disk I/O with central processor activity. We can replace transitions a_4, a_5, and location $b_8[3]$ of Figure 5 by the net shown in Figure 6 (the definition of this modification can be expressed in the same manner as illustrated previously, but will not be given here). This implies that another attribute for disk time is necessary, which determines the transition time for a_{11}. The transition time for a_5 would become zero, and $t(a_{13})$ is determined by trace data carried in $M(b_{19}(3))$.



b_9: Job through with CP and disk
b_{14}: Job ready to use disk and CP
b_{15}: Disk is idle
b_{16}: Job is requesting disk
b_{17}: Disk is busy
b_{18}: Job is through with disk
b_{19}: CP is busy
b_{20}: Job is through with CP
b_{21}: Job ready to relinquish CP

Figure 6—Parallel central processor and disk activity

SUMMARY

The class of evaluation nets has been informally described. An evaluation net may be treated as an *interpreted marked directed graph*, where transitions correspond to vertices and the locations correspond to directed arcs. The arcs are capable of holding a single item of structured data at a time. The graph of the net represents the structure of the system and indicates the control of token flow. The transition procedures interpret the action of the vertices. By operating the net, (in a simulation manner), measures of resource utilization, turnaround, throughput, etc., are available for further analysis of the system. An implementation of the nets might include some "automatic" analysis, such as resource utilization figures. The nets are modular and allow varying level of detail of representation. An interactive implementation of evaluation nets might consist of a net editor with graphic and symbolic output. The graphic output would be used by the designer in structural debugging and the symbolic output could be used by an interpreter to simulate the net. Current studies at the University of Washington include the implementation of evaluation nets.

A more formal treatment of the nets may be found in Nutt,[7] from which this paper is abstracted. Examples are given which model the Boolean functions of two variables and a Turing machine. A comprehensive evaluation net of the CDC 6400 is presented which shows the structure of the machine and which allows an extensive performance evaluation of the machine at the task/resource level. This net includes models of priority queues of arbitrary length and illustrates how queueing algorithms may be handled. Evaluation nets are also compared with Petri nets.

Future work, besides the implementation, includes the study of the nets as models for computational processes.

ACKNOWLEDGMENT

The author is grateful to Jerre D. Noe for his support of the research and to Alan C. Shaw for his editorial suggestions.

REFERENCES

1 H C LUCAS JR
Performance evaluation and monitoring
Computing Surveys 3 No 4 pp 79-91 1971

2 C A PETRI
Kommunikation mit automaten
PhD dissertation University of Bonn 1962
Translated by C F Greene Jr Applied Data
Research Inc Technical Report No RADC-TR-65-377 1
supl 1 1966

3 J D NOE
A Petri net description of the CDC 6400
Proceedings of ACM Workshop on System Performance Evaluation
Harvard University pp 362-378 1971

4 A W HOLT et al
Information system theory report
Applied Data Research Inc Technical Report No
RADC-TR-68-305 1968

5 A W HOLT F COMMONER
Events and conditions
Record of the Project MAC Conference on Concurrent
Systems and Parallel Computation pp 3-52 1970

6 J D NOE G J NUTT
Validation of a trace-driven CDC 6400 simulation
SJCC Proceedings Volume 40 pp 749-757 1972

7 G J NUTT
The formulation and application of evaluation nets
PhD dissertation University of Washington Computer
Science 1972

Objectives and problems in simulating computers

by THOMAS E. BELL

The Rand Corporation
Santa Monica, California

INTRODUCTION

Because the effort required to simulate a computer system is often very great, analysts should consider carefully the probable value of the results prior to embarking on it. Special languages[1-5] have been created to aid the programmer in reducing the time required to code a simulation, and analysis techniques[6-11] are available to reduce time requirements in the later phases of a study. Still, unexpected problems usually arise: An effort concludes with a study only partly completed because budgeted resources have been exhausted,* or the results may be of less value than anticipated. If the analyst can foresee problems prior to commencing the detailed coding phase of a study, he can avoid many of the problems, mitigate many of the remainder, and allow for the rest in anticipating the payoffs of the effort.

While some of the problems encountered have unique characteristics, a common set of them seems to keep appearing in simulation studies of computers. Simply knowing the total list of all common problems is no solution to the analyst who typically goes over budget; his difficulty is sorting out the problems that are most relevant to his situation and ignoring the rest. Trying to plan for the unlikely and unimportant can deflect effort from more appropriate areas and lead to less effective analysis than would occur if the problems were ignored until they appeared. The objectives of the simulation influence how the situation will be approached and which problems will most likely lead to critical difficulties.

The challenge facing the analyst is to associate the potential problems with his objectives so that he can anticipate his most probable pitfalls and allocate his resources to solving these problems. He needs a list of problems, a list of objectives, and, finally, a matrix showing which objectives lead to which problems. With this information he can plan his effort more effectively* and improve the design of his simulation model.

SIMULATION PROBLEMS

Problems in simulating computer systems could be organized into (1) choosing the language for the simulation, (2) representing the real system appropriately, (3) debugging the simulation, (4) performing experiments, and (5) interpreting the results.** This classification scheme jumps to the analyst's mind immediately because, chronologically, these are the steps he takes in performing a simulation analysis. Although procedural frameworks are important and may lead to improved simulations, they usually do not attempt to identify which particular issues will be most important for a specific simulation effort throughout the procedure.

For example, the analyst, in designing his simulation, must consider the resources available to him and how flexible his work must be. He can choose his simulation language by considering these and several other issues. The underlying problems he encounters in language choice and the other steps in a study amount to resolv-

* See Reference 2, page 2.

* One of the most important advantages in the planning stage is an ability to predict the costs and specific payoffs of an effort. Overselling the potential payoffs of a simulation not only puts the actual results in question, but decreases the credibility of future simulations.

** A more useful scheme is suggested by Morris (Chairman of the Association for Computing Machinery's Special Interest Group on Simulation) and Mayhan in an unpublished paper:[12] (1) Define the problem; (2) select a solution method; (3) develop models; (4) validate models; (5) simulate alternative solutions; (6) select and implement the best alternative; and (7) validate simulation solutions.

ing them correctly. Some of the most troublesome are the following:*

1. Resources. The amount of manpower and machine resources to perform a simulation study may be greater than the expected value of the study, or they may simply exceed the total resources available for the effort. The desire should always be to minimize invested resources, but the characteristics of some simulations make this issue more critical than in other studies. (The total available resources may be very limited—particularly in terms of elapsed time—and the challenge very great.) Typically, adequate resources are invested in the early phases of an effort with the later phases receiving whatever is available. The issue of resources is mentioned on page 150 of Reference 14.
2. Changes. Changes to improve model validity, to produce additional output, and to reflect modified objectives can prove a major difficulty in some simulations, while they are relatively trivial in others. Although some simulation efforts are not complicated by unexpected changes, quick examination of simulation code often reveals that changes were far more extensive than anticipated. Inadequate appreciation of the need for change can lead to choosing a language that is too inflexible as well as designing code that is too complex. The need for changes in a model is noted on page 87 of Reference 15.
3. Boundaries. In addition to changes as described above, a simulation analyst may find that the boundaries defining the modeled portion of the system change as the study progresses. He may discover that he has attempted to simulate too much of the system and be forced to replace parts of the simulation with simple functions. Alternatively, he may find that his boundaries are too narrow, and important interactions are not being reflected. Identifying the degree to which boundaries will need change can alter a simulation's design to reduce the difficulty of boundary redefinition. Dumas[16] refers to the problem of boundaries on page 77 of his paper.
4. Costs. Cost models are often of significant utility, particularly when the objective includes analysis for procurement or performance improvement decisions. Their inclusion, however, often implies a heavier investment of resources in order to determine the costs of purchasing hardware or software. Costs of using alternative systems (including costs of delays) often prove particularly difficult to quantify. The importance of cost models is noted in References 17 and 18.
5. Experimental design. Toward the end of many simulation efforts analysts realize that exercising the simulation will not be a straightforward process. At this late date, they begin to consider how to design experiments: Are 500 hours of CPU time adequate to determine the response surface? Many documents deal with the problem, including References 6 and 8–11.
6. Detail. Simulations vary in detail of implementation from those that are relatively gross (References 19 and 20 give examples) to those that represent operations at the micro-instruction level (References 21 and 22 present examples). The level of detail can often be expressed as the smallest increment of time explicitly recognized in the model. If the simulation is performed in a language like GPSS[23] or CSS,[24] this level is explicitly recognized in the language. However, this indicator of minimum time increment, although quantitative, conceals the essence of the problem, which is to decide on the extent that system interactions are to be replicated.
7. Accuracy. Analysts should always desire to have the ultimately achievable degree of accuracy in a simulation as high as possible. However, the utility of improved accuracy may be very low and hardly worth the cost. This issue is addressed in References 19 and 25.
8. Validation. An analyst's belief in the accuracy of his simulation is inadequate for evaluating its actual closeness to reality. Only a formal validation effort can reduce the doubt that it is unrepresentative of the real system. The degree of representativeness is usually assumed to be definable by the ability of the simulation to produce a few numbers that are close to the numbers obtained from reality. Other types of validity are often important, however, including correct sequences of operations and correct responses to alterations of input. The analyst must determine the most appropriate degree of effort to be expended in validating his model. Although many simulations of computers are never validated, examples of validation exercises can be found in References 19 and 25.

* Few authors even mention the problems they have encountered in simulating a computer system; this may explain the impression held by some that such efforts are easy. References to sources dealing with specific issues are given in the descriptions of the issues. McCredie and Schlesinger[13] mention nearly every one of the issues in their paper.

OBJECTIVES

The objectives of a simulation should be explicitly stated and should be closely related to the decision environment in both terminology and emphasis. Simulation for its own sake is a sterile process and economically unjustifiable. Some published papers on specific simulations state that the author's objective was to simulate a particular jobstream on a particular hardware/software system. These papers probably reflect the author's orientation toward the problems involved in the simulation activity per se; the decision-environment objectives can usually be deduced from sections titled "Findings" or "Conclusions." Five categories of simulation objectives seem to characterize the bulk of simulations of computer systems. These five categories are as follows:

1. Feasibility analysis—investigating the possibility of performing a conceptualized workload on a general class of computer systems. An example of a feasibility analysis is presented in Reference 26.
2. Procurement decision-making—comparing one or more computer systems with a specific workload to decide which of several (or whether any) computer systems should be procured. For example, Bell Telephone Laboratories reports this type of simulation application in Reference 27, and page 4 of Reference 28 provides a report of Mobil Oil Corporation's application.
3. Design support—projecting the effects of various design decisions and/or tracking the development of a system. Many simulations have design as the objective. Examples are to be found in References 15, 16, and 29.
4. Determining capacity—for projected systems, determining the processing capacity of various configurations; for existing systems, determining the processing capacity of a load different from the current work. Examples of what were apparently simulations to determine capacity are presented in References 30 and 31.
5. Improving system performance—increasing processing capacity by identifying and changing the most sensitive parts of the hardware/software system. This process is also known as tuning, and examples can be found in References 20 25, and 32.

Decision-oriented objectives may be as hard to state at the beginning of an effort as they are to discover in many post-analysis papers. Nevertheless, analysts somehow manage to choose an approach and then develop some solution to each of the issues suggested earlier. Many of these are developed within the context of other choices (e.g., the language to be used) involving additional, mechanistic criteria (e.g., user-directed output). One danger in using this procedure is that the process of making other choices may seriously compromise the simulation's value by directing the simulation into unfruitful areas.

ISSUE	OBJECTIVE				
	Feasibility	Procurement	Design	Determining Capacity	Improving System
Resources					
Changes					
Boundaries					
Costs					
Experimental Design					
Detail					
Accuracy					
Validation					

Figure 1—Desired matrix

Just as importantly, the analyst may attempt to generate a simulation that will do all possible things. McCredie and Schlesinger* point out that attempting simulations "capable of answering almost any reasonable question about the system...must be paid for by large investments in personnel and computer time." This is true, of course, because the analyst must solve all the problems indicated earlier, and some of these may have solutions for one objective that are inconsistent with solutions for other objectives.

Such inconsistent solutions should be detectable by drawing a matrix of the issues and objectives with the general solutions as entries. Figure 1 illustrates such a matrix, but it is not completed because the objectives are not well enough defined to permit identification of even a general solution for each issue. For example, the most appropriate level of detail in a study to determine the feasibility of computer logic might be at the microinstruction level as it is in Rummer's study.[33] At the same time, a simulation to investigate the feasibility of an entire system might be at so high a level that nothing shorter than a complete job task or data transmission is considered. (This is the case in the studies by Downs, et al.[26] and Katz.[34]) Yet both simulations would have

* Pages 201-202 of Reference 13.

feasibility as the objective. A different categorization scheme is needed for objectives—one that will make it easier to associate problems with objectives by aggregating the decision environment's objectives into classes for the simulation environment.

Alternative categorization scheme for objectives

The alternative scheme suggested in this paper does not divide the objectives into more categories; instead, it reduces the number and redirects them so that they are more useful in defining answers to the issues suggested above. The alternative defines three categories: absolute projection, sensitivity analysis, and diagnostic investigation. It may appear that all the simulations in each of the decision-oriented five categories map easily, as blocks, into categories in the alternative scheme of three, but exceptions appear often enough that generalizations about mappings are dangerous.

Absolute projection

This category includes those simulation studies whose objectives can be reduced to the desire to make basically dissimilar comparisons. An example of this type of objective is a situation in which the processing capacities of two systems under a certain load are to be compared. The analyst wishes to determine which system should be procured. Another example is the comparison of a system's processing capacity with the load that it is expected to encounter. (This is usually tested operationally by determining the expected time for the simulated system to process a load and comparing this time with the maximum allowable time.) The decision under consideration in this instance may be whether to procure a certain system or it may be whether to perform a new job on an existing system. A third example of a simulation in this category is one in which response times are being compared with stated requirements. If the proposed system is unable to meet the requirements, then it must be augmented.

The important characteristic in each of these examples is the necessity for evaluating an objective function in absolute terms with a high degree of absolute accuracy. If two systems actually differ in processing capacity by 20 percent, the simulation technique must produce answers with absolute errors of less than 10 percent if the analyst is to be sure of choosing the better system.

Apparent examples of absolute projection are described in References 14, 26, and 30.

Sensitivity analysis

Simulations falling in this category emphasize similar comparisons. While simulation studies making absolute projections must have absolute accuracy, sensitivity analyses require good accuracy only in (1) the areas in which two cases are not identical and (2) the areas that significantly interact with the nonidentical areas. Although the simulation code may represent far more than the portion of the system under consideration, the primary validation effort should be devoted to the central portion, with reasonableness being the criterion for the rest. The remainder of the simulation code is seldom excess (and therefore an embarrassment) for several reasons. First, it usually interacts with the central portion in some manner in which the details are not important, but the general sorts of interaction are important. Second, other sensitivity analyses may use the same simulation code, and building one simulation for several analyses may be the most efficient procedure. Third, the boundaries of the central portion are often not identifiable early in the simulation effort because the analyst is not yet familiar with all the interactions.

A decision-maker doing sensitivity analysis may require that answers have high reliability, but if he has an alternative that improves on the default by 20 percent, he does not need to have the absolute values of each. His decision is based on the changed value of the objective function rather than its absolute value.

A basic characteristic of sensitivity analysis, of course, is the comparison of slightly different alternatives. For the simulation analyst this implies that his simulation must be constructed to facilitate changes.

As an example of sensitivity analysis, an analyst might be interested in the effects of changing hardware, changing software, or changing scheduling schemes. Specifically, he might want to know whether increasing the size of buffers results in increased message throughput. With the exception of the changes under consideration, the initial and changed simulations are identical. The analyst must ensure that the changed portion (and the parts it interacts with) are represented accurately, but the remainder of the simulation (probably including disk queuing, front-end processors, file layouts, etc.) can be less accurate. Of course, the possibility exists that the analyst will incorrectly assume that parts of the system are not critical when they really are, but this is the boundary problem that an analyst always faces. He might find comfort in having the simulation agree with reality in correctly reporting message throughput over a wide range of conditions, but his decision can be made on the basis of the ratio of throughputs before and after the increase in message size.

References 16, 18, 21, 22, 29, and 35 would appear to give examples of sensitivity analysis.

Diagnostic investigation

Diagnostic investigations tend to place less emphasis on the value of an objective function. The interest of the analyst is to gain understanding of the detailed manner in which the simulated system behaves. He may be interested in examining interactions, in analyzing aberrations in the real system (or, too often, those peculiar to his simulation), or in tracing the progress of a transaction to determine whether it goes through the system as expected. The emphasis tends to be on performance of very small parts of the system. Graphical analysis techniques often find application in this type of simulation since detailed sequences of activities may require examination.

Diagnostic investigation would appear to be the objective in References 32 and 33.

Substudy objectives

The global objectives of a simulation study may not match the immediate objectives of an analyst at certain points in a study. For example, an analyst performing a sensitivity analysis study may find that he needs to project absolute performance to determine whether his model's gross interactions yield results that are even remotely correct. Then he may wish to verify the details of an alternative scheduling strategy and trace its actions through the scheduling algorithm. Only then does he bother to perform simulation runs for the several alternatives that he has programmed. Although his global objective would fall in the category of sensitivity analysis, the analyst would have performed two substudies with local objectives in the other two categories of absolute projection and diagnostic investigation. Dumas[16] and Ceci and Dangel[36] have performed these types of substudies.

The substudy objectives in a simulation study constitute a means of attaining the study's objectives and merely reflect short term techniques. While these may be important in performing tasks such as verification and ensuring reasonableness, they are not the objectives that determine the simulation's overall design and should not confuse the analyst about the type of global objectives he is pursuing. If the effort devoted to a substudy becomes large, the analyst should carefully consider whether his substudy effort is relevant, his formal global objectives should be revised, or the global objectives are simply unattainable.

ASSOCIATING PROBLEMS WITH OBJECTIVES

The definitions of problems and objectives suggested in the preceding pages have assumed that the analyst is interested in designing his simulation before launching into the details of language choice and coding. The assertion has been implicit that these definitions could be used in associating the problems with the objectives to lead to better simulation designs. Figure 2 represents an attempt to provide this type of aid. The importance of the first five issues (resources, changes, boundaries, costs, and experimental design) are indicated there; the applicability of most of the entries is apparent.

For example, limitations on available resources will be a critical problem in absolute projection studies because the entire system must be simulated to a high degree of accuracy, and usually the work must be done in a short time. On the other hand, a diagnostic investigation need only reflect particular parts of the system of interest. Sensitivity analyses lie somewhere between these two extremes.

Suggestions regarding the last three issues are of a different character. Rather than indicating importance (largely the degree of resource commitment needed), they suggest approaches that are not necessarily indicative of a particular level of effort; however, taking the right action is critical for a simulation of any objective.

Level of detail

The most appropriate level of detail for an absolute projection simulation is usually at quite a macro level because the entire system must be simulated, and resources are usually at a premium. At the other extreme, a diagnostic investigation usually must be at a relatively micro level in order to reflect detailed interactions. A sensitivity analysis simulation, however, often represents a combination of levels since it may represent the bulk of the system grossly and the altered part in detail.

Accuracy

The accuracy of response time or throughput figures in a diagnostic investigation study is usually of superficial importance. The analyst is investigating the manners in which one (or a few) parts of the system interact; investigating the details of one part of a system's behavior does not require overall accuracy of performance parameters. Of course, the performance should be reasonable or the behavior will not be reasonable, but

ISSUE	OBJECTIVE		
	Absolute Projection	Sensitivity Analysis	Diagnostic Investigation
Resources	Critical	Important	Desirable
Changes	Desirable	Critical	Important
Boundaries	Desirable	Important	Critical
Costs	Desirable	Important	Irrelevant
Experimental Design	Important	Critical	Desirable
Detail	Macro	Moderate	Micro
Accuracy	Critical Overall	Critical in Places	Reasonableness Only
Validation	Value Comparison	Derivative Comparison	Sequence Checking

Figure 2—Issues vs objectives

high accuracy in performance parameters is not necessary for representative interactions.

For sensitivity analysis, a simulation must closely reflect the differences that will be encountered between the various alternatives under consideration. While absolute values of performance parameters may be comforting, the decision problem at hand requires only the relative difference between similar situations.

Absolute projection, of course, requires accuracy in desired performance parameters; undue faith in absolute projections is perhaps the most dangerous error in simulating computer performance.

Validation

Validation is performed to improve the confidence that the required type and degree of accuracy is obtained in a simulation. This means that, in absolute projection, the simulation's projection of performance parameters must be compared with the parameters from the real system. This value comparison is necessary if faith is to be vested in the predicted parameters. In instances where a system does not exist (so no validation can be performed), the analyst should include a caveat with any reported results to indicate that the simulation is of undetermined accuracy.

Sensitivity analyses are often validated by comparing the values of real performance parameters with the predicted values over some set of conditions that are realizable on the actual system being simulated. A projection can then be made of an unvalidated case based on the knowledge that the predicted performance was correct in a number of similar cases; therefore, the changes in performance were accurately reflected and probably will be in the new case. This approach may be excessively expensive because it requires accuracy in parts of the simulation that are not to be altered. An alternative is to compare only changes in performance due to specific changes in the system. In this case, only the fractional changes need be compared, so significant savings may be possible. This less exhaustive process is analogous to comparing derivatives rather than absolute values of functions. In many cases, the analyst only needs to determine whether it is positive, negative, or zero.

Validation of diagnostic investigations requires even less rigor than for sensitivity analyses. Since the emphasis is on examining detailed interactions, the analyst usually only needs to ensure that the sequence of operations is correct. Even this validation can be quite time-consuming and frustrating if the analyst is restricted to viewing flowcharts. Powerful graphical techniques for showing interactions are very useful here.

APPLICATION

The categorization schemes and matrix presented in this paper are without value unless they can aid analysts in planing analyses and designing simulations. Two examples will be given to indicate how they can be applied. One example uses a simulation that was performed without reference to such schemes and illustrates how the effort could have been aided by their use. (The problems encountered led to developing the schemes and matrix.) The second example presents a situation in which the schemes were applied in order to avoid problems that otherwise might have arisen.

Example 1: Simulating without reference to the matrix

This first example involves a simulation of the Video Graphics System (VGS) performed during the implementation of software on newly designed hardware. The system uses a central communications switching and controlling machine—an IBM 1800—that communicates with a series of terminals and several service machines. The service machines execute user code and send digital representations of pictures to the 1800 for conversion to analog representation in a special picture generator controlled by the 1800. One picture generator and three scan converters service all users (presently 28) who employ terminals with raster scanned screens that can be slightly modified broadcast television sets. Various input devices, including keyboards, are added to the sets to enable two-way communications. The objective of the system is to supply high-powered interactive graphics capability to many users at a moderately low cost through time-shared use of the expensive digital-to-picture hardware. The system as a whole is described in Reference 37 and a description of the modeled portion of the system is presented in Reference 38.

Objectives

Prior to doing any simulation coding, we spent time learning about the system's characteristics and developing a set of simulation objectives. We then distributed a preliminary description of our understanding of the system along with our proposed objectives. (The objectives were expressed as questions that needed answers.) The characters to the left of each objective did not appear in the original (taken from Page 80 of Reference 38). They indicate the type of objective, and the characters stand for the following:

A	Absolute projection.
S	Sensitivity analysis.
D	Diagnostic investigation.

Although several additional objectives were added during the study, the objectives listed below were retained for its duration. Many of these objectives, however, were not addressed due to lack of time and the belief that the questions could not be adequately answered with the simulation.

A 1. Under what load conditions will the system give poor response? (It may be feasible to alter the load by user education as well as by changing characteristics of such software support as the Integrated Graphics System.)

A 2. Will messages be unduly delayed in the VMH* system in the 360s?

A 3. Will channel cycle-stealing slow the 1800 CPU enough that input data are lost due to delays in processing?

S 4. Will a ping-pong system decrease response time of the VGS?

D 5. What will be the effect of the 1800 waiting at interrupt level four while buffers are unavailable for service machine input?

D 6. What will be the effect on the 1800 of one service machine being unresponsive for a short period?

A 7. What portion of system capacity does a Tablet take? (It might be profitable to disable a Tablet that is temporarily not in use, or to use a keyboard instead of the Tablet.)

S 8. How useful would more core be in the 1800?

S 9. How useful would another 1800 be?

These objectives included four in the category of absolute projection, three in sensitivity analysis, and two in diagnostic investigation. Since we had objectives in each of the three objective categories, we can see from Figure 2 that we needed a simulation that was at a macro level but also (conflicting) in micro detail. In addition, the simulation had to have a high degree of overall accuracy, be easy to change, have easily altered boundaries, use few resources, and be extensively validated. While no one noted the extreme difficulty of achieving all the objectives at the time we stated them, our proposed categorization schemes and matrix of solutions quickly reveals how difficult it would be to achieve them all.

Diagnostic investigations

We decided to create a simulation at a low level of detail; the basic time increment in this GPSS simulation was 50 microseconds. It traced all normal interactions in the 1800 and used approximate timing information generated by multiplying the number of instructions in a module by the average time per instruction as measured during an early run of the system. Most of the actual work with the simulation involved diagnostic investigations, including objectives 5 and 6. In addition to the objectives stated before coding began, we investigated cases of potential deadlock and the platooning that were characteristic of the system. Interactive computer graphics was used extensively to aid in investigating these situations; hardcopy graphics was used to document the results and communicate with system designers. Figure 3 shows a typical output, complete with the analyst's marginal notes. This display shows, over simulated time, the priority level of the executing software at the top; the entire bottom of the display presents a Gantt chart. This Gantt chart shows which routines are in control at each moment of simulated time. With these displays the simulation served admirably in answering questions during diagnostic investigations.

* Video Message Handler, essentially an access method.

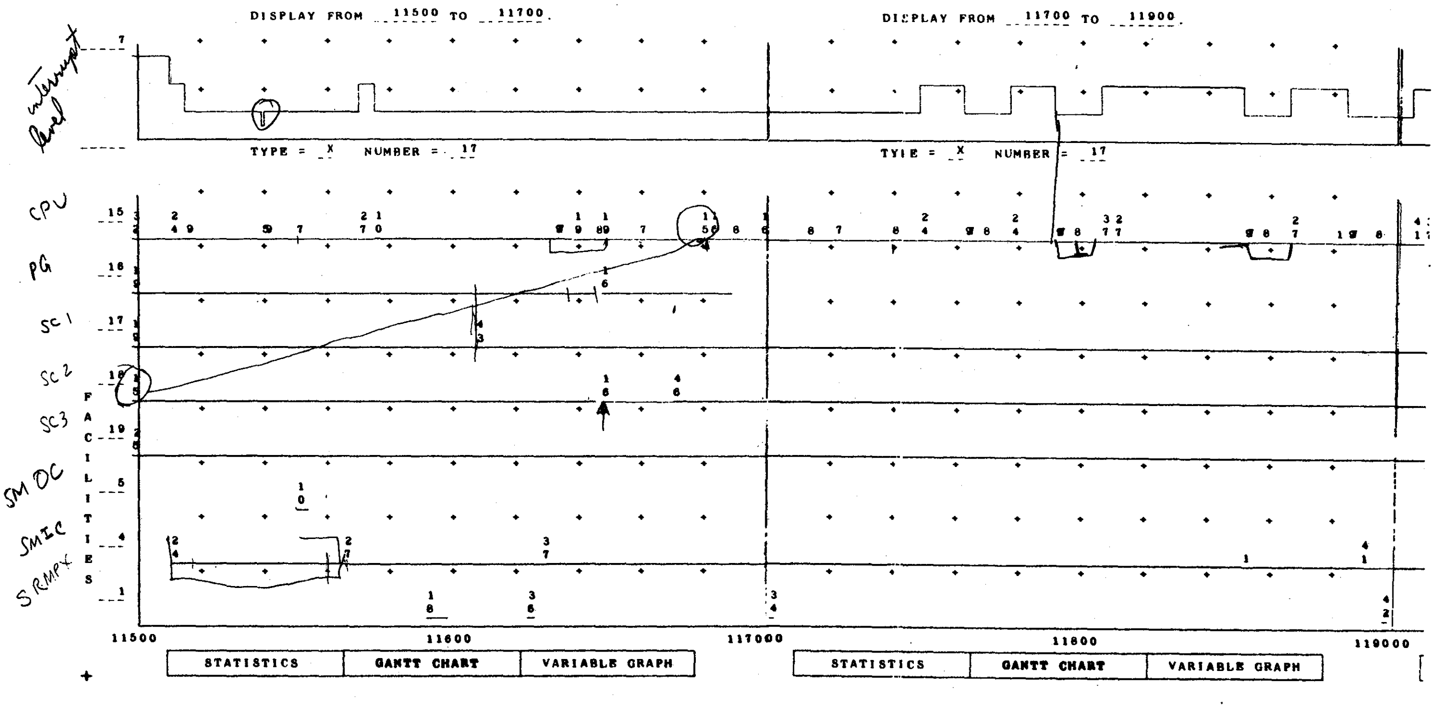


Figure 3—Graphical output

Sensitivity analysis

The initial sensitivity analysis objectives were addressed, but delays in validation caused us to be very reluctant to put much faith in the results. Early use of the real system indicated that some functions performed by hardware should be implemented in software, and we decided to add an objective about the utility of this change. We found, to our surprise, that total system utilization would be only marginally affected by the change. Without validation, we discounted this result initially. The importance of the issue, however, led to a substudy with strong characteristics of diagnostic investigation to explain the result. We discovered that low priority attempts by the system to clean up various queues caused processing in the altered case (with hardware implementation) but not in the initial case (with a software implementation). Eventually, we did perform a validation of the simulation and found that, within the context of sensitivity analysis, the simulation had quite adequate accuracy. (See Reference 39 for details of this validation effort.)

Absolute projection

The largest number of objectives for this simulation study fell into the category of absolute projection. Objective 7 (regarding the portion of system capacity used by a single RAND Tablet) is typical of these, and illustrates the problems of using a simulation for absolute projection when it is designed to fulfill other objectives too. The first problem is that the portion of system required by a Tablet varies with system loading. As the load increases, the overhead to handle a Tablet (contrary to usual system performance) decreases. Therefore, a single number is inadequate to represent performance in general. This characteristic of systems (performance not being easily represented in simple ways) appears in most systems, but, in absolute projection, stating the fact is often considered unacceptable by people desiring simple answers.

Secondly, the absolute projections, in comparison with measured results, tended to be optimistic by a factor of about two. That is, reality took twice as long as predicted by the simulation. Since projections were based on average instruction times, we put the 1800 processor into a very restricted processing loop (83 instructions) to separate timing assumptions from interaction representations, computed the predicted time to execute the instructions (using published, manufacturer supplied timings), and measured the actual time to execute them. In a variety of instances the actual and predicted did not agree; in one of the clearest cases the prediction was 155 microseconds and the measured time was 220 microseconds. This last difference led us to doubt that our bottom up approach to generating timings would ever lead to fulfilling most of the absolute projection objectives since we did not understand some of the interactions between hardware and software.

(One of the few objectives that were usefully addressed, even if not rigorously answered, was objective 7. The predicted system loading was so high that even gross errors in the simulation would not lead to acceptable performance. Predicting this performance problem helped strengthen the case for hardware implementation of some of the functions necessary for Tablet operation.)

Summary

Many of the anticipated payoffs of the simulation were not realized because its objectives implied inconsistent solutions to problems in simulation. While the results were useful in fulfilling some objectives, a review of the problems to be encountered in achieving the other objectives could have allowed us to rank them and to consider, before coding the simulation, whether its design was most appropriate in aiding the VGS designers.

Example 2: Referencing the matrix before simulating

This second example involves a simulation of a very large information management system. The simulation was undertaken during the design stage; no hardware was yet available for running any validation tests. A "packaged simulator" was to be used to determine the size of hardware to be ordered. The objective clearly fell into the absolute projection category, and yet, validation of program descriptions could not be performed. While management wished to know the precise configuration that should be acquired, facilities were not available for performing the necessary validations of overall accuracy.

Diagnostic investigation

We suggested that diagnostic investigations be undertaken to determine whether some critical portions of software would perform as expected. The micro-level simulations could be checked for correct sequences and, as soon as hardware was available, the reasonableness of the predictions could be validated.

Absolute projection

The need for information about appropriate hardware configurations was very real, so we suggested that a multi-phase strategy be pursued. During the period when no validation was possible, important programs could be simulated at a macro level to see whether obvious design problems existed. (If the simulation predicted 100 hours to run each of ten daily programs, even the most skeptical analyst would question the design. A number of such instances were discovered and corrected.) The important element of this phase was to devote heavy effort only to cases where problems clearly exceeded the potential errors in the simulation. Concurrently, techniques for describing programs were checked by employing them on software being run on an existing system. This effort led to changes in the descriptions of software for use in simulations. Later, preliminary validation could be performed using data made available from configurations used in testing. Since analysts had already completed initial simulations of the programs, validation and revision could be accomplished in the short time between availability of initial data and the required hardware ordering date.

Summary

Suggestions about a more appropriate procedure for this example could clearly be made without our schemes and matrix. In practice, however, they often are forgotten in the rush to implement something and show results. Further, opinion about the difficulty of a specific task is a weak tool to use in convincing people who are unfamiliar with simulation's limitations or under heavy pressure to "get on with the job." The categorization schemes and matrix of solutions are convenient techniques for indicating the requirements to achieve a certain objective in comparison with other potential objectives.

RECOMMENDATIONS

We have found the application of this approach useful in planning and designing our own simulations and in helping other analysts to improve theirs. It proves particularly useful in predicting how much effort is appropriate for validation exercises and what form such validation should take. While an experienced simulation analyst may feel that it expresses little that he does not already know, too many analysts fail to apply their knowledge rigorously in the early stages of a simulation effort.

We suggest that analysts force themselves to state objectives clearly—and in writing—at the beginning of a simulation effort. They should then consider whether all their objectives are realizable when using the suggested solutions to the problems listed in the matrix of

Figure 2. Only after assuring themselves that the effort can result in fulfilling the objectives should they design the simulation. Finally, they should consider whether the achievement of the objectives will justify the cost required to implement and validate the simulation.

REFERENCES

1 L J COHEN
S3 The system and software simulator
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 282-285

2 N R NIELSEN
ECSS: An extendable computer system simulator
The Rand Corporation RM-6132-NASA February 1970

3 J N BAIRSTOW
A review of system evaluation packages
Computer Decisions Vol 2 No 6 June 1970 p 20

4 W C THOMPSON
The application of simulation in computer system design and optimization
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 286-290

5 G K HUTCHINSON J N MAGUIRE
Computer systems design and analysis through simulation
Proceedings AFIPS 1965 Fall Joint Computer Conference Part 1 pp 161-167

6 G S FISHMAN
Estimating reliability in simulation experiments
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 6-10

7 T E BELL
Computer graphics for simulation problem-solving
Third Conference on Applications of Simulation ACM et al New York December 1969 pp 47-56 (Also available as RM-6112 The Rand Corporation December 1969)

8 D P GAVER JR
Statistical methods for improving simulation efficiency
Third Conference on Applications of Simulation ACM et al New York December 1969 pp 38-46

9 T H NAYLOR K WERTZ T H WONNACOTT
Methods for analyzing data from computer simulation experiments
Communications of the ACM Vol 10 No 11 November 1967 pp 703-710

10 G S FISHMAN
Problems in the statistical analysis of computer simulation experiments: the comparison of means and the length of sample records
The Rand Corporation RM-4880-PR February 1967

11 G A MIHRAM
An efficient procedure for locating the optimal simular response
Fourth Conference on Applications of Simulation ACM et al New York December 1970 pp 154-161

12 M F MORRIS A J MAYHAN
Simulation as a process
Simuletter Vol 4 No 1 October 1972 pp 10-15

13 J W McCREDIE S J SCHLESINGER
A modular simulation of TSS/360
Fourth Conference on Applications of Simulation ACM et al New York December 1970 pp 201-206

14 H A ANDERSON
Simulation of the time-varying load on future remote-access immediate-response computer systems
Third Conference on Applications of Simulation ACM et al New York December 1969 pp 142-164

15 A L FRANK
The use of simulation in the design of information systems
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 87-88

16 K DUMAS
The effects of program segmentation on job completion times in a multiprocessor computing system
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 77-78

17 S R CLARK T A ROURKE
A simulation study of cost of delays in computer systems
Fourth Conference on Applications of Simulation ACM et al New York December 1970 pp 195-200

18 N R NIELSEN
An analysis of some time-sharing techniques
Communications of the ACM Vol 14 No 2 February 1971 pp 79-90

19 J D NOE G J NUTT
Validation of a trace-driven CDC 6400 simulation
Proceedings AFIPS 1972 Spring Joint Computer Conference Vol 40 1972 pp 749-757

20 J H KATZ
Simulation of a multiprocessor computer system
Proceedings AFIPS 1966 Spring Joint Computer Conference Vol 28 pp 127-157

21 S C CATANIA
The effects of input/output activity on the average instruction time of a real-time computer system
Third Conference on Applications of Simulation ACM et al New York December 1969 pp 105-113

22 S E McAULAY
Jobstream simulation using a channel multiprogramming feature
Fourth Conference on Applications of Simulation ACM et al New York December 1970 pp 190-194

23 *General purpose simulation system/360 user's manual*
H20-0326 International Business Machines Corporation White Plains New York 1967

24 *Computer system simulator II (CSS II) general information manual*
GH20-0874 International Business Machines Corporation White Plains New York 1970

25 P E BARKER H K WATSON
Calibrating the simulation model of the IBM system/360 time sharing system
Third Conference on Applications of Simulation ACM et al New York December 1969 pp 130-137

26 H R DOWNS N R NIELSEN E T WATANABE
Simulation of the ILLIAC IV—B6500 real-time computing system
Fourth Conference on Applications of Simulation ACM et al New York 1970 pp 207-212

27 J M JENKINS R G MAHER
Uses of simulation in the design of large scale information systems
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 85-86

28 R A CANNING
Data processing planning via simulation
EDP Analyzer Vol 6 No 4 April 1968 13 pp

29 M H MacDOUGALL
Simulation of an ECS-based operating system
Proceedings AFIPS 1967 Spring Joint Computer Conference Vol 30 pp 735-741

30 L C SANDERS
A Monte Carlo process for determining response times for tactical systems
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 79-84

31 W I STANLEY H F HERTEL
Statistics gathering and simulation for the Apollo real-time operating system
IBM Systems Journal Vol 7 No 2 1967 pp 85-102

32 M M LEHMAN J L ROSENFELD
Performance of a simulated multiprogramming system
Proceedings AFIPS 1968 Fall Joint Computer Conference Vol 33 Part 2 pp 1431-1442

33 D I RUMMER
FORTRAN simulation of digital logic
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 297-305

34 J H KATZ
An experimental model of system/360
Communications of the ACM Vol 10 No 11 November 1967 pp 694-702

35 S L REHMANN S G GANGWERE JR
A simulation study of resource management in a time-sharing system
Proceedings AFIPS 1968 Fall Joint Computer Conference Vol 33 Part 2 pp 1411-1430

36 R J CECI G W DANGEL
On-line system simulation
Digest of the Second Conference on Applications of Simulation ACM et al New York December 1968 pp 89-93

37 K W UNCAPHER
The Rand video graphics system—an approach to a general user-computer graphics communication system
The Rand Corporation R-753-ARPA April 1971

38 T E BELL
Modeling the video graphics system: procedure and model description
The Rand Corporation R-519-PR December 1970

39 T E BELL
Computer performance analysis: minicomputer-based hardware monitoring
The Rand Corporation R-696-PR June 1972

A methodology for computer model building

by A. DeCEGAMA

The National Cash Register Company
San Diego, California

INTRODUCTION

System performance evaluation techniques are of vital importance in the system design process. As depicted schematically in Figure 1, the selection of the design variables is generally accomplished by an iterative process in which the evaluation of the system cost and performance plays a crucial part.

Simulation and mathematical modelling constitute the two basic approaches to computer system performance evaluation. Simulation models can be built to study almost any system with a very fine degree of detail, but they may require an inordinate amount of time for the determination of the system performance. On the other hand, mathematical models, while being more limited in scope, are much faster than simulation models and consequently, much more economical to apply. This fact gives mathematical models a decisive advantage. When they can be built, mathematical models of complex systems with large numbers of design variables can be used to optimize the designs; whereas, the number of simulation runs required to accomplish the same task would be so high that the simulation approach to computer system optimization is totally impractical.

But good and realistic mathematical models of computer systems are difficult to develop. The real world is far too complex to be described faithfully with a series of equations. Therefore, approximations requiring extensive testing, guided by a deep understanding of the stochastic processes involved, must be made in order to formulate a model that, while capturing the essence of the problem at hand, is mathematically and computationally feasible.

Furthermore, mathematical models of computer systems are costly to develop because their verification requires in turn the development and application of very detailed simulation models that are as close to the real systems as possible. The cost involved in a project of this type may well run into the hundreds of thousands of dollars, due to the number and lengths of the simulation runs involved, which require large and fast machines.

These two stumbling blocks: the difficulty of building a good model and the cost of verifying it, are the main reasons why good mathematical models of computer systems are practically nonexistent.

This paper presents a methodology to build mathematical models of multiprogramming systems that can greatly reduce the time and cost involved in developing models for specific systems.

THE CPU INTERRUPT PROCESS IN MULTIPROGRAMMING SYSTEMS

The occurrence of CPU interrupts in a multiprogramming system is the result of the superposition of many random and independent events (paging, I/O file accesses, ends of programs, time-slicing, spooling, etc.) that are caused by the concurrent processing of different programs. This situation is illustrated in Figure 2 which represents the succession in time of the interrupts produced by five programs residing simultaneously in main memory. Programs 1 and 2 are assumed to be of priority 1 (the highest), program 3 of priority 2 and programs 4 and 5 of priority 3.

The Pooled Output Theorem[1] states that the distribution of the CPU inter-interrupt times (Ti) would be asymptotically exponential, as the number of programs in main memory increases, if the individual inter-interrupt times within any priority i, T_{ii}, were independent.

The individual T_{ii} are the result of a sum of random variables:

$$T_{ii}=T_{io}+T_{wi}+T_{ui}+T_o$$

where

T_{i0} = I/O response time/interrupt (includes both waiting and service time)
T_{wi} = CPU waiting time for programs of priority i
T_{ui} = CPU service time between consecutive interrupts for programs of priority i
T_0 = CPU interrupt overhead

It can be seen that there is some interdependence between the different T_{ii} due to the I/O and CPU waiting times involved. But, since in a system properly designed the CPU and I/O service times should be much longer than the corresponding waiting times, such interaction may not be strong enough to constitute a significant deviation from the condition of independent T_{ii}.

Also, since the trend toward the exponential density is very rapid with an increasing number of programs in memory, it would appear that a finite number of programs being multiprogrammed provide conditions that may sufficiently approach those for which the Pooled Output Theorem is applicable. This situation is reinforced by the fact that, in addition to the interrupts caused by the programs residing in main memory at any given time, the CPU must also be interrupted for the control of the input and output (spooling) processes. Since the input and output processes are truly independent, their superimposition with the interrupt process due to the programs being simultaneously served in main memory results in an even faster trend toward the exponential density for the overall system interrupts.

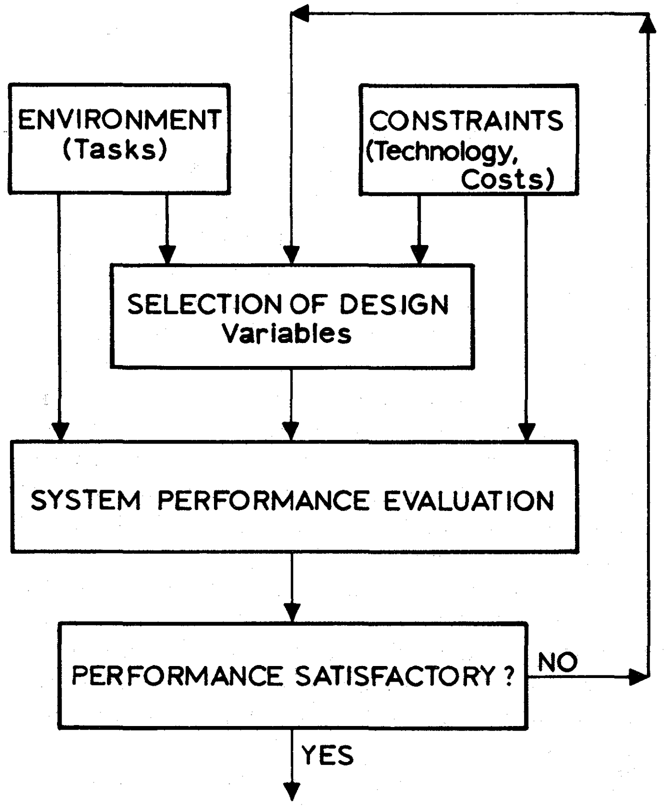

ENVIRONMENT (Tasks)
CONSTRAINTS (Technology, Costs)
SELECTION OF DESIGN Variables
SYSTEM PERFORMANCE EVALUATION
PERFORMANCE SATISFACTORY ?
NO
YES

Figure 1—Schematic representation of the system design process

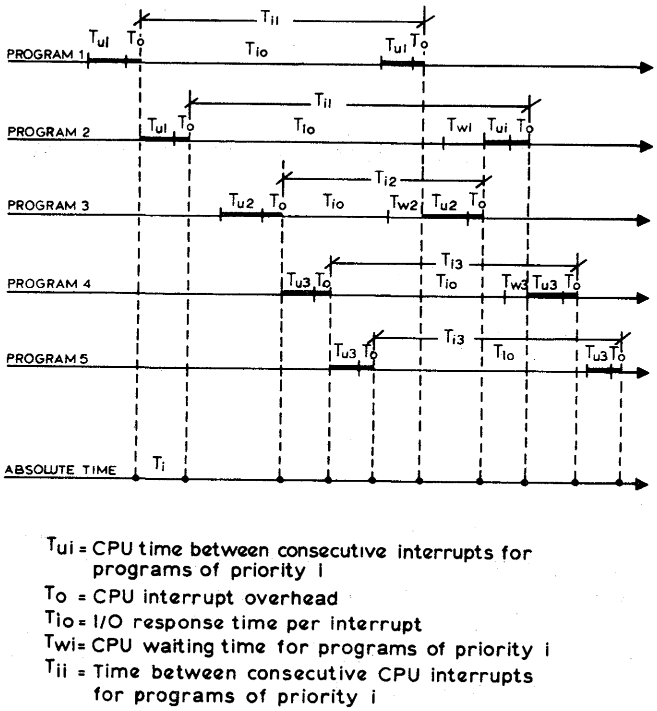

PROGRAM 1
PROGRAM 2
PROGRAM 3
PROGRAM 4
PROGRAM 5
ABSOLUTE TIME
Tii
Tui
To
Tio
Twi
Ti2
Ti3
Tu2
Tw2
Tu3
Tw3
Ti
Tui = CPU time between consecutive interrupts for programs of priority i
To = CPU interrupt overhead
Tio = I/O response time per interrupt
Twi = CPU waiting time for programs of priority i
Tii = Time between consecutive CPU interrupts for programs of priority i

Figure 2—Times between consecutive CPU interrupts (T_i) caused by the programs residing in memory in a multiprogramming environment

If the applicability of the Pooled Output Theorem can be verified, it can have important implications for the development of mathematical models of multiprogramming systems, since their formulation basically consists of the analysis of complex stochastic service systems in which several queueing processes take place simultaneously. Such queueing processes can be analyzed mathematically only if the interarrival times between consecutive service requests to each of the different system elements (CPU, disk, drum, etc.) are exponentially distributed. This can only happen if the CPU interrupts constitute a Poisson process.

STATISTICAL ANALYSIS OF THE INTER-INTERRUPT TIMES IN MULTIPROGRAMMING SYSTEMS

A SIMSCRIPT Simulation Program was developed with the purpose of investigating the intrinsic stochastic characteristics of the interrupt process in multiprogramming systems.

The design of the simulation model was sufficiently flexible to permit the study of the effect of diverse environment and system characteristics in the inter-interrupt times.

The variables that could be varied between simulation runs and their corresponding ranges of variation are shown in Table I.

The forms of the distribution functions of the CPU and I/O service times and interarrival times could also be changed. In addition, the CPU and I/O service disciplines could be defined at the beginning of a run to be either First-Come-First-Served (FCFS) or by priorities. In the latter case, a choice could be specified between Pre-Emptive Resume and Non-Pre-Emptive disciplines.

A Simulation run ended after 10,000 interrupts had been generated. The output of each simulation run consisted of statistics for equipment utilization and queue lengths and distribution functions for CPU holding times, I/O service times, waiting times, and inter-interrupt times. The individual inter-interrupt times corresponding to the last 1,001 interrupts (1,000 values) were written on a tape. It was assumed that this sample was representative of the steady-state of the simulated system.

After each simulation run, a statistical analysis of the values generated on the simulation output tape was performed.

TABLE I—Variables Changed Between Simulation Runs to Investigate The Interrupt Process in Multiprogramming Systems

Variable	Range
No. programs being multiprogrammed	4-10
No. priorities	1-3
Type of I/O devices	Drum-Disk-Tape
Distribution of I/O service times	10-100 msec (avg)
Distribution of CPU service times between consecutive I/O interrupts	5-10,000 msec (avg)
CPU Quantum size	50-500 msec (∞ in the cases where there was no quantum interrupt)
Distribution of program inter-arrival times	.3-10 secs. (avg)
No. CPU interrupts for input/program	5-100 (avg)
No. CPU interrupts for output/program (spooling)	5-100 (avg)

The statistical analysis consisted of the following tests:

1. Tests to ascertain the stationarity of the series of inter-interrupt times (no time trend).
2. Tests to detect serial correlation between successive inter-interrupt times (if the inter-interrupt times are stationary with no serial correlation, they constitute a renewal process. The Poisson process with its exponentially distributed inter-event times is a special type of renewal process).
3. Distribution-free tests of goodness of fit for the Poisson process:
   (a) One-sided Kolmogorov-Smirnov Statistic
   (b) Two-sided Kolmogorov-Smirnov Statistic
   (c) Anderson-Darling Statistic
4. Specific tests of the Poisson Hypothesis against renewal hypotheses:
   Moran Statistic (most powerful test against a renewal alternative in which the intervals have a Gamma density)
5. Tests to measure deviations from the exponential distribution:
   (a) Maximum deviation from the exponential distribution with the same average as the studied series.
   (b) Coefficient of variation (the quotient of the standard deviation and the average which is equal to 1 for the exponential distribution).

Even though it may appear that the number of tests is excessive, this statistical analysis is actually a condensed version of more extensive existing procedures.[2] Such procedures usually require a large number of tests for the Poisson hypothesis due to the lack of a single test with satisfactory consistency and power against a wide range of alternatives. The indicated tests are assumed to provide sufficient complementarity, so that the conclusions drawn can be applied with a reasonable degree of confidence.

Alternate tests for exponentiality based on chi-square methods[3] were contemplated but were not applied due to their low power.

The rationale behind the indicated procedure is that, if the simulator generated data appear to be consistent with an assumed underlying exponential distribution at a given significance level, then the deviation between the actual underlying distribution and the assumed exponential distribution with the same average value should be tolerable. The tests to measure the deviations

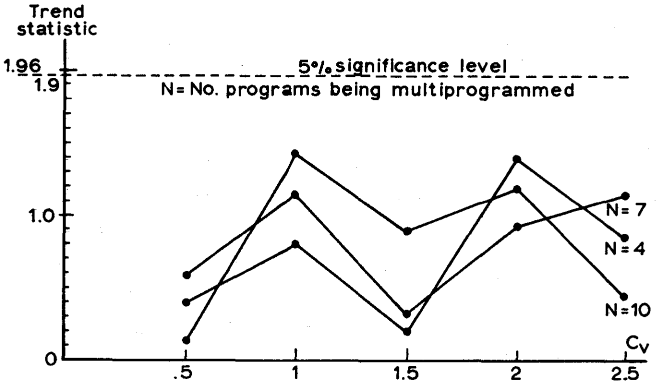


Figure 3—Trend of inter-interrupt times vs coefficient of variation of basic CPU+I/O cycle

from the exponential distribution are intended to determine the validity of this rationale.

The main results obtained by simulation followed by the statistical analysis are shown in Figures 3 through 6. The statistics in those figures are presented as a function of N, the number of programs being multiprogrammed and C_v, the coefficient of variation of the corresponding program service cycles, defined as the sum of the CPU service time plus the I/O service time between consecutive interrupts for an individual program.

It was observed during the course of this investigation that the closeness of the CPU interrupt process to a Poisson process in a multiprogramming system is just a function of these two parameters alone. In other words, for a given value of N, all combinations of program and system characteristics resulting in the same values for C_v give similar values for each statistic considered. Thus, C_v is a convenient parameter to identify different multiprogramming environments from the standpoint of the interrupt characteristics of individual programs.

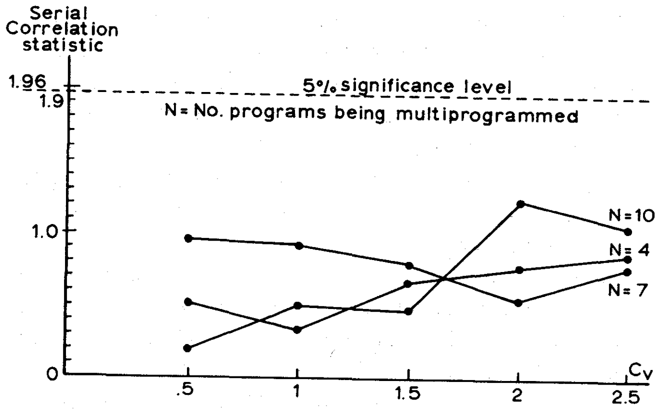


Figure 4—Serial correlation vs coefficient of variation of basic CPU+I/O cycle

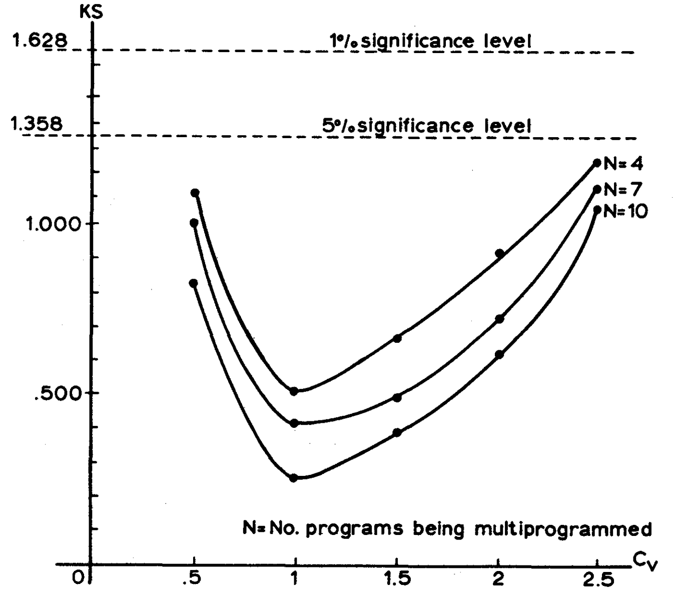


Figure 5—Two-sided Kolmogorov-Smirnov statistic vs coefficient of variation of basic CPU+I/O cycle

As was to be expected from the Pooled Output Theorem and the properties of the Poisson process, the higher the number of programs being multiprogrammed and the closer C_v is to 1, the stronger the indications are that the CPU inter-interrupt times are exponentially distributed.

The points of Figures 3 through 6 represent average values of each statistic considered. The values of a given

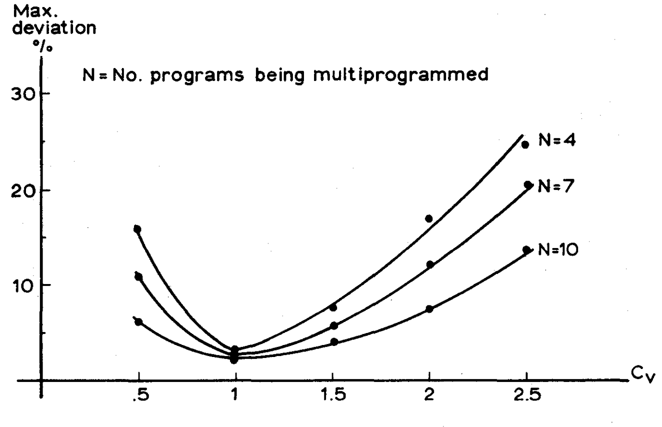


Figure 6—Maximum deviation (%) from exponential density vs coefficient of variation of basic CPU+I/O cycle (frequency interval 10%)

statistic characterized by the same N and C_v are closely clustered around the indicated average point. Thus, it can be said, based on experimental evidence, that only two parameters, N and C_v, are needed to determine whether the exponential hypothesis can be applied in any given multiprogramming situation.

The conclusions that can be drawn from the simulations and statistical analyses are:

1. The interrupt process in a multiprogramming system constitutes most probably a renewal process (Figures 3 and 4 show clearly the stationarity and very low serial correlation of the process).
2. The deviation of the interrupt process in a multiprogramming system from the Poisson process is a function of the number of programs being multiprogrammed and the coefficient of variation of the CPU-I/O service cycle exclusively (Figures 5 and 6 show trends that are typical of all the computed statistics).
3. There was no outright rejection of the Poisson hypothesis at the 5 percent significance level. This was true even for the Moran Statistic which is a most powerful test. The implication is then that the assumption of an underlying exponential distribution is not inconsistent with the observed inter-interrupt times.
4. The maximum deviation from the exponential distribution that has been observed (Figure 6) in any sample appears to be tolerable. This is borne out by the results obtained with computer models based on the exponential hypothesis. As later indicated, the application to the models of queueing theory expressions requiring exact exponential distributions yielded only small errors in all the cases considered.
5. The range of applicability of the exponential hypothesis is at least

   $$N \geq 4$$
   $$.5 \leq C_v \leq 2.5$$

   This range comprises all the studied cases which included most practical multiprogramming situations.

MODELING IMPLICATIONS

The basic fact that the occurrence of interrupts in a multiprogramming system constitutes a Poisson-like process for a wide range of programming and system characteristics is the keystone for a powerful technique of computer model building.

In any system where the inter-interrupt times are exponentially distributed, the interarrival times of service requests to the different I/O devices are also exponentially distributed. This is due to the fact that selecting events at random with a given probability p from a Poisson process results in another Poisson process. If the density function of the times between events in the original process is $\lambda e^{-\lambda t}$, the corresponding density function in the derived process has the same form but with parameter λp instead of λ. In a given system the probabilities of requiring access to the different I/O devices can be calculated as a function of measured program and system characteristics.[4]

If the interarrival times to the I/O devices are exponentially distributed, then the average and variance of the I/O response time from each device can be determined by standard Queueing Theory formulas as a function of the average interarrival time and the first three moments of the corresponding service time.[4] No knowledge is required of the actual forms of the service time distributions for a majority of service disciplines. The first three moments of the service times are easily calculated[4] from the measured first three moments of the amount of data to be transferred and the access and transfer characteristics of the different I/O devices.

The possibility of calculating accurately the average and variance of the I/O response time per interrupt leads directly to a methodology to develop realistic mathematical models of multiprogramming systems.

MODEL BUILDING METHODOLOGY

The main objective of any mathematical model of a computer system is the determination of the system performance for any configuration and programming environment.

The key to the determination of a system's performance is the calculation of the first two moments of the service time/program for the given hardware, software and programming characteristics. This makes possible the computation of the two fundamental measures of system performance: throughput or the quantity of service provided per unit time and the average response time per program that indicates the quality of the service.

The system throughput is equal to the program arrival rate, if it can be sustained by the system. This is determined by comparing the average number of programs that can be concurrently serviced with the actual average number of programs that must be multiprogrammed.

The number of programs that the system can serve simultaneously is either a fixed constant or it is a function of the memory and program sizes and the storage allocation algorithm.

On the other hand, the average number of programs that must be processed concurrently is equal to the quotient of the average service time per program and the average program interarrival time. If this value is not less than the number of programs that can be multiprogrammed, then the assumed program arrival rate cannot be sustained and it must be reduced.

With respect to the average response time per program, it can be calculated as a function of the average program interarrival time and the first two moments of the service time per program only if the program interarrival times are exponentially distributed. The reason for this is that the response time per program is the sum of the service time and the time waiting in the queue of programs trying to enter main memory to begin service. Mathematical expressions for the average value of the waiting time only exist for the case of exponentially distributed program interarrival times when the service time distribution is of general form.

This limitation is not as serious as it may seem. In addition to system performance prediction, the most important application of a mathematical model of a computer system is as a basic component in a system optimization program. If the target function for optimization is the maximization of throughput, the form of the program interarrival time distribution is not important. If the target function is the minimization of the average response time, its calculation is not needed. This can be seen by considering that the phenomenon of waiting is a direct consequence of the variances of the arrival and service processes. The variance of the program interarrival times is uncontrollable but the variance of the service times can be minimized to achieve the least possible value of the average waiting time.

Thus, the determination of the first two moments of the service time/program as a function of the system and programming environment characteristics constitutes the basic calculations of the mathematical model of a computing system.

Service time/program

The service time/program can be expressed as

$$T_s = T_{cp} + T_{\mathrm{Io}}$$

where

T_s = Service time/program
T_{cp} = CPU time/program (service time plus waiting time)
T_{Io} = I/O time/program (service time plus waiting time)

Consequently, the average and variance of the service time/program are calculated by

$$E[T_s] = E[T_{cp}] + E[T_{\mathrm{Io}}]$$

and

$$\mathrm{Var}\,[T_s] = \mathrm{Var}\,[T_{cp}] + \mathrm{Var}\,[T_{\mathrm{Io}}]$$

respectively.

T_{cp} is in turn equal to the sum of a numberof components:

$$T_{cp} = T_{pw} + T_{pe} + T_{po}$$

where

T_{pw} = Time waiting for CPU service/program
T_{pe} = CPU execution time/program
T_{po} = CPU overhead time/program

The average and variance of T_{cp} are then given by

$$E[T_{cp}] = E[T_{pw}] + E[T_{pe}] + E[T_{po}]$$

and

$$\mathrm{Var}\,[T_{cp}] = \mathrm{Var}\,[T_{pw}] + \mathrm{Var}\,[T_{pe}] + \mathrm{Var}\,[T_{po}]$$

I/O time/program

T_{Io} is equal to the sum of a random number of random variables:

$$T_{\mathrm{Io}} = \sum_{j=1}^{k} \sum_{1}^{N_j} T_j$$

where

k = number of I/O devices in the system
N_j = number of interrupts/program resulting in access to devices of type j
T_j = Response time for devices of type j

The average value of T_{Io} is calculated by

$$E[T_{\mathrm{Io}}] = \sum_{j=1}^{k} E[N_j]E[T_j]$$

and the variance by

$$\mathrm{Var}\,[T_{\mathrm{Io}}] = \sum_{j=1}^{k} (E[N_j]\,\mathrm{Var}\,[T_j] + \mathrm{Var}\,[N_j]E^2[T_j])$$

according to standard Probability Theory expressions for the sum of a random number of random variables.[5]

As has been explained, the average and variance of

T_j can be determined only because the service request interarrival times are quasiexponentially distributed.

With respect to N_j, its average and variance can be calculated by applying the indicated expressions for the sum of a random number of random variables to the program execution time, i.e.,

$$E[T_{pe}] = E[N_j]\,\frac{E[T_{ij}]}{P_{ij}}\,(1-P_{ij})$$

$$\mathrm{Var}\,[T_{pe}] = E[N_j]\left(\frac{\mathrm{Var}\,[T_{ij}]}{P_{ij}}\,(1-P_{ij}) + \frac{E^2[T_{ij}]}{P_{ij}^2}\,(1-P_{ij})\right) + \mathrm{Var}\,[N_j]\,\frac{E^2[T_{ij}]}{P_{ij}^2}\,(1-P_{ij})^2$$

where

T_{ij} = CPU time between events (execution of jump instructions, data references, file references, etc.) that may result in an I/O interrupt to access a device of type j

P_{ij} = Probability of actually accessing a device of type j

and where the actual CPU time between interrupts is again the sum of a random number of random variables. The random variables are the T_{ij} and their random number between interrupts has a negative binomial distribution with average

$$\frac{1-P_{ij}}{P_{ij}} \quad \text{and variance} \quad \frac{1-P_{ij}}{P_{ij}^2}$$

Solving for $E[N_j]$ and Var $[N_j]$

$$E[N_j] = \frac{E[T_{pe}]P_{ij}}{E[T_{ij}](1-P_{ij})}$$

$$\mathrm{Var}\,[N_j] = \left[\mathrm{Var}\,[T_{pe}] - \frac{E[T_{pe}]P_{ij}}{E[T_{ij}](1-P_{ij})} \times \left(\frac{\mathrm{Var}\,[T_{ij}](1-P_{ij})}{P_{ij}} + \frac{E^2[T_{ij}]}{P_{ij}^2}\,(1-P_{ij})\right)\right]\frac{P_{ij}^2}{(1-P_{ij})^2E^2(T_{ij})}$$

$E[T_{pe}]$ and Var $[T_{pe}]$ as well as $E[T_{ij}]$ and Var $[T_{ij}]$ must be measured for each environmental program mix. P_{ij} should be calculated as a function of system architectural features such as cache memories design parameters and system resource management features such as paging algorithms and buffering schemes, etc. The computations can become very involved[4] and they are beyond the scope and length of this paper. Suffice it to say that a number of good models exist[6,7,8,9,10] that can be applied to compute the different P_{ij}.

CPU time/program

With respect to the remaining components of T_{cp} not yet determined, the average and variance of T_{po}, the interrupt overhead per program, and T_{pw}, the time waiting for CPU service per program, are calculated by considering that T_{po} and T_{pw} are sums of random numbers of random variables:

$$E[T_{po}] = E[T_{oi}]\sum_{j=1}^{k} E[N_j]$$

$$\mathrm{Var}\,[T_{po}] = \mathrm{Var}\,[T_{oi}]\sum_{j=1}^{k} E[N_j] + E^2[T_{oi}]\sum_{j=1}^{k}\mathrm{Var}\,[N_j]$$

$$E[T_{pw}] = E[T_{ws}]\sum_{j=1}^{k} E[N_j]$$

$$\mathrm{Var}\,[T_{pw}] = \mathrm{Var}\,[T_{ws}]\sum_{j=1}^{k} E[N_j] + E^2[T_{ws}]\sum_{j=1}^{k}\mathrm{Var}\,[N_j]$$

where

T_{oi} = CPU overhead/interrupt

T_{ws} = Time waiting for CPU service between consecutive CPU services

The average and variance of T_{oi} must be measured. The average and variance of T_{ws} must be calculated. This is accomplished by considering the CPU as a stochastic service system similar to the I/O devices.

The process of requesting CPU service is analogous to the CPU interrupt process and is also Poisson-like (Figure 2). In other words, the interarrival times of CPU service requests can be considered to be exponentially distributed with the same average as the inter-interrupt times. Simulation shows that this approximation is also very close to reality.[4]

In addition to the average time between CPU service requests, the first moment of the CPU holding times is required. It is simply

$$E[T_{ch}] = \left(\sum_{j=1}^{k}\frac{P_{ij}}{E[T_{ij}](1-P_{ij})}\right)^{-1} + T_{oi}$$

and, since the different T_{ij} are truly independent, it is reasonable to expect that the CPU holding times resulting from their superposition have a density function that tends toward the exponential density as k increases. Simulation shows[4] that in systems with several levels of secondary storage, the approximation is as close as that for the inter-interrupt times. Under those conditions, the variance of the CPU holding times is just equal to the average squared. Also, the CPU waiting times are exponentially distributed if the interarrival and holding times are both exponentially distributed and consequently $\mathrm{Var}[T_{ws}]=E^2[T_{ws}]$. $E[T_{ws}]$ is calculated by standard Queueing Theory expressions depending on the service discipline as a function of the average inter-interrupt time and the average CPU holding time.

System interrupt rate

The calculation of the average inter-interrupt time completes the backwards presentation of the basic steps that must be taken to build a model (when building an actual model of a system, development proceeds step by step from interrupt rate toward the computation of performance).

The average inter-interrupt time is given by

$$E[T_i]=E[T_{ch}]/U_{cp}$$

where U_{cp} represents the CPU utilization factor which is calculated by

$$U_{cp}=(E[T_{pe}]+E[T_{po}])Q_s$$

where Q_s indicates the system throughput in programs per unit time.

Equipment utilization constraints

In addition to the indicated calculations to build a system's model, a set of constraints must be simultaneously satisfied in order for the system to be able to maintain the desired rate of throughput. These are the constraints for equipment utilization: CPU, Main Memory and I/O devices that must be all less than one. How to compute the CPU utilization factor has already been indicated. The main memory utilization can be computed as

$$Q_sE[T_s]/N_s$$

where

N_s = average number of programs that can reside in main memory and receive service simultaneously (multiprogrammed)

The I/O device utilization is simply the quotient of the average service time and the average service request interarrival time.

The preceding outline of the steps to be taken to build a model of a multiprogramming system can be applied to interrelate system variables (hardware and software), programming variables and resource management variables. (A list of typical environmental, system and controllable variables has been published elsewhere.[4,16])

Thus, a model built by applying the described methodology will relate intimately all these variables in a set of equations that constitute the mathematical expressions of the basic interrelationships of the system. Therefore, the resulting model can be used for optimization purposes, since the effect of the change of any one variable on system performance can be readily calculated by the model.

MODULAR DESIGN

The described model building approach is susceptible to modular implementation. As depicted in Figure 7, the determination of the system interrupt rate is the focal point of this methodology. The steps to be taken to determine it as a function of the system configuration and characteristics and how to use it to obtain the system performance have already been outlined. The point to be made here is that many specialized computations that can be modelled separately are also required to build a complete model of a computer system.

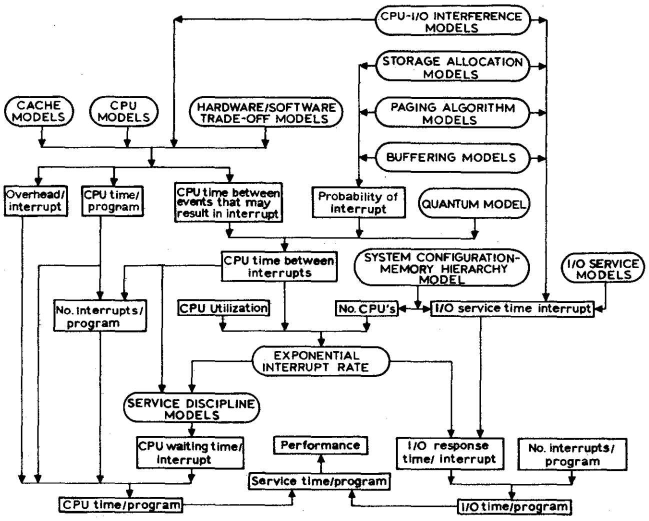

CPU-I/O INTERFERENCE MODELS
STORAGE ALLOCATION MODELS
CACHE MODELS
CPU MODELS
HARDWARE/SOFTWARE TRADE-OFF MODELS
PAGING ALGORITHM MODELS
BUFFERING MODELS
Overhead/ interrupt
CPU time/ program
CPU time between events that may result in interrupt
Probability of interrupt
QUANTUM MODEL
CPU time between interrupts
SYSTEM CONFIGURATION-MEMORY HIERARCHY MODEL
I/O SERVICE MODELS
No. interrupts/ program
CPU Utilization
No. CPU's
I/O service time interrupt
EXPONENTIAL INTERRUPT RATE
SERVICE DISCIPLINE MODELS
CPU waiting time/ interrupt
Performance
I/O response time/ interrupt
No. interrupts/ program
Service time/program
CPU time/program
I/O time/program

Figure 7—Basic model building methodology and modular design

For instance, a Paging Model and a Storage Allocation Model are needed to determine the probability that a paging interrupt will occur when a memory address is generated by the CPU.

A System Configuration Model is required to specify the hierarchy of memories in the system and Buffering Models are required to determine the number and size of the I/O buffer areas and the probabilities of actually having to perform an I/O operation when a READ or WRITE instruction is executed. If the CPU service is quantized, a Quantum Interrupt Model is necessary to obtain the actual average CPU time between interrupts.

Also, it must be kept in mind that the CPU times between consecutive events of the same type that may result in an interrupt and that are assumed to be known, have either been measured in the system under study or in some other system. In the latter case, several more models may be needed to determine the corresponding CPU times based on the characteristics of the new system. In the first place, a CPU Model that can compare different CPU and memory designs and instruction sets and determine the relative CPU powers to process a given type of task is required to convert the CPU time measurements to the new system. Since the interference with other CPU's and I/O Modules may slow down the CPU and lengthen the CPU times required for a given task, a CPU-I/O module Interference Model must also be applied. In addition, if the system under study has a cache memory, a Cache Model is needed to determine the impact of the cache on the program processing speed of the CPU. And, if possible hardware/software trade-offs are being studied, another model is required to establish their effect on the interrupt rate and the distribution of I/O accesses.

In addition to the indicated models, a variety of I/O service models for different types of devices and service disciplines are also needed to be able to apply the basic Model Building Methodology depicted schematically in Figure 7 to a variety of situations.

Thus, a substantial number of functional models are necessary to implement a complete model of a multiprogramming system. The degree of detail of the functional models determines the complexity of the overall system model and its accuracy. Priorities of programs can be considered, and distinctions of processing requirements between compilations and executions or code and data types of information can be made.[4] The linking of the functional models that results in a unified system model is provided by those parameters and variables that are common to one or more functions. For instance, some of the variables of the Storage Allocation Model (page size, number of pages allowed in memory, program sizes) that influence system performance are also required in combination with the System Configuration Model to determine the structure of the hierarchy of memories in the system.

But this interdependence alone is not sufficient to make the overall system model feasible. Without the knowledge that the system interrupt rate constitutes a Poisson process, the entire model structure would collapse. A basic link between the functional modules would be missing. Then, what makes this Model Building Methodology very powerful is the fact that the individual functional models can be developed and verified independently from one another. Thus, a given functional model can be built by applying new mathematical developments, by experimenting with simulations or by regression analysis. The important point is that the problem of developing a complex model of the overall system has been considerably simplified and reduced to manageable proportions.

VERIFICATION

Mathematical models of multiprogramming systems developed by applying the described methodology have been verified by comparing their predictions with the results of a very detailed simulation model of multiprogramming systems. The simulation model has been written in GPSS and it has great flexibility to simulate different system configurations, service disciplines and control policies. In order to give an indication of how closely the results of the two models follow one another, Table II presents the calculated by a mathematical model and the measured by simulation values for optimum performance of some of the important parameters that correspond to some of the systems to which the models have been applied. The systems investigated had three levels of storage (main memory, bulk memory (ECS), disk). Their environments are indicated in Table II.

The industrial environment consists of three priorities of programs: conversational, real-time and batch. In the university environment there are three priorities of programs also, but instead of the real-time system there is a computer aided instructional system requiring on the average, more CPU time but less memory space than the real-time system.

An optimization procedure based on the mathematical model and using Direct Search Techniques[11] was applied in all cases. Table II shows clearly the value of mathematical models to predict system performance. The closeness of the results of the mathematical and simulation models seems to indicate that good models have been developed.

TABLE II—Comparison of Results of the Mathematical and Simulation Models

	Parameter 1		Parameter 2		Parameter 3		Parameter 4		Parameter 5		Parameter 6	
CASE	First moment of CPU time between interrupts for priority 3 (msec.)		Average interarrival time to disk controller (msec)		CPU utilization		Average service time for priority 1 (msec)		Average service time for priority 2 (sec)		Average service time for priority 3 (sec)	
1	Calc.	Meas.	Calc.	Meas.	Calc.	Meas.	Calc.	Meas.	Calc.	Meas.	Calc.	Meas.
University environment; programs and data accessed in main memory.	5.79	5.25	147	165	.44	.444	507.6	464.1	42.9	41.57	188.3	190
2 University environment; user programs and data accessed directly in ECS (Extended Core Storage).	181.7	184.1	127	151	.722	.73	575.1	559.8	47.66	50.16	272.33	277.38
3 University environment; priorities 1 and 2 accessed in main memory; code and data of priority 3 accessed directly in ECS.	19.46	18.6	108	121.5	.54	.544	207.1	189.87	27.7	25.4	306.39	311.54
4 Business environment; programs and data accessed in main memory.	10.1	11.5	129	151	.362	.366	221.15	210.2	2.21	2.19	154	153.67
5 Business environment; programs and data accessed in ECS (user programs).	186.14	187.6	96.16	107	.548	.551	393	374	2.71	2.45	166.33	174.24

The general problem of verification of simulation models has been extensively discussed in the literature.[12,13,14,15] It has been said[15] that "in view of the difficulty which arises in attempting to agree upon a set of criteria for establishing when a model is verified, efforts should concentrate on the degree of confirmation of a model rather than whether or not the model has been verified." It is felt that all the numerous models that have been built according to the methodology presented in this paper have been confirmed to a high degree.

CAPABILITIES

Mathematical models of computer systems relate all the important parameters and variables of the hardware, software and environment to the indexes of performance and cost by means of computations that can be carried out very quickly. Consequently, they provide a flexible and efficient way for the study, design and control of computer systems. Some typical applications of models developed along the lines discussed in this paper are:

Optimum hardware configurations for given applications

The cost/performance gains that can be obtained by system optimization can be significant. Improvements of up to 30 percent of the cost/performance index were observed[4] in the optimization of systems whose controllable variables were initially chosen by an intuitive process of educated guess.

Cost-performance evaluations for hardware-software trade-offs

A mathematical model of a multiprogrammed business computer has already been applied to study the improvement in system performance that can be obtained by a Firmware Sort-Processor.[16]

Evaluation of new system architectures based on advanced technologies

A mathematical model of a cache has been used in conjunction with an overall system model[17] to reveal some interesting facts about the effectiveness of cache memories to improve system performance.

Adaptive operating system design

If the programming environment of a system and its hardware availability can be measured dynamically, then it is possible in principle to have some kind of control over the allocation of resources and the scheduling of tasks so that a high level of performance is maintained under dynamically changing conditions. The ability of a mathematical model of a system to predict its performance for any combination of environmental parameters and controllable variables makes it the focal point of an algorithm for system performance optimization. It is suggested that such an algorithm can provide an operating system of conventional design with an effective mechanism for decision making.

MEASUREMENTS

The environmental measurements that must be made for the application of mathematical models of multiprogramming systems are: measurements of times between certain events that may result in an interrupt, measurements of amounts of space required, measurements of distances between data or instruction addresses referenced by the programs and frequencies of usage of the different elements of system and user programs.

The cost of gathering and analyzing the required data should be offset by the improvements in performance and cost savings that can be achieved through the application of mathematical models.

SUMMARY

A new methodology to design mathematical models of computer systems has been introduced. The hypothesis of exponential inter-interrupt times is the cornerstone of this new approach to computer model building. The range of validity of this hypothesis has been specified. It is demonstrated that a diversity of models can be built based on this concept and on a modular functional approach to model development. The main computational steps to be followed in order to link together partial functional modules and build an overall system model have been specified. Simulation has shown the validity of the exponential hypothesis in many multiprogramming situations and the basic soundness of models developed by applying the defined methodology. Some typical applications for which mathematical models are particularly suited have been indicated.

ACKNOWLEDGMENTS

The encouragement and constructive criticism provided by Mr. Harut Barsamian of the NCR-DPD Research Department is sincerely appreciated.

The author is also grateful to Mrs. Lori Lehman and Mrs. MaryAnn Garrison for their contributions in typing the paper.

REFERENCES

1 D R COX P A LEWIS
The statistical analysis of series of events
John Wiley & Sons Inc New York 1966

2 P W W LEWIS T C KELLY
A computer program for the statistical analysis of series of events
IBM Research Report RJ 362 November 12 1965

3 E M SCHEUER
Testing grouped data for exponentiality
RAND Memorandum RM 5692 PR August 1968

4 A DeCEGAMA
Performance optimization of multiprogramming systems
Doctoral Dissertation Computer Science Department
Carnegie Mellon University Pittsburgh Pa April 1970

5 W FELLER
An introduction to probability theory and its applications
Vol I John Wiley & Sons Inc New York 1968

6 P J DENNING
The working set model for program behavior
Communications of the ACM May 1968

7 E GELENBE J C A BOEKHORST
J L W KESSELS
Minimizing wasted space in partitioned segmentation
Proceedings of the Symposium on Computers and Automata
Polytechnic Institute of Brooklyn New York April 1971

8 E G COFFMAN JR T A RYAN
A study of storage partitioning using a mathematical model of locality
Communications of the ACM March 1972

9 P J DENNING
Properties of the working set model
Communications of the ACM March 1972

10 A WOOLF
Analysis and optimization of multiprogrammed computer systems using storage hierarchies
University of Michigan Technical Report RADC TR 71 165
August 1971

11 C F WOOD
Recent developments in direct search techniques
Research Report 62 159 522 R1 Westinghouse Research
Laboratories Pittsburgh Pa.

12 D K TOCHER
The art of simulation
The English Universities Press Ltd 1963

13 *General purpose simulation system 360 user s manual*
IBM Publication H20 0326 0 January 1968

14 T H NAYLOR K WERTZ T WONNACOTT
Some methods for analyzing data generated by computer simulation experiments
National Meeting of the Institute of Management Sciences
Boston April 1967

15 T H NAYLOR J M FINGER
Verification of computer simulation models
Management Science October 1967

16 H BARSAMIAN A DeCEGAMA
Evaluation of hardware firmware software trade offs with mathematical modeling
AFIPS 1971 SJCC Proceedings

17 H BARSAMIAN A DeCEGAMA
Some design considerations of cache memories
IEEE Compcon 1972 Proceedings San Francisco

LOGOS and the software engineer*

by C. W. ROSE

Case Western Reserve University
Cleveland, Ohio

INTRODUCTION

Most of us consider a well-engineered product to be one which is structurally sound; which communicates with its environment in a predictable, well-disciplined manner; which has been thoroughly tested; and which is reliable and easily maintained. In any engineering field, the structural philosophy, design disciplines, and checkout methods which yield such a product are called "good engineering practices." Software engineering is the application of good engineering practice to the design, implementation and final checkout of large programs. The result of effective software engineering should be:

(1) The production of a correct program (certifiable)
(2) The availability of means of efficiently determining the correctness of a program (certification)
(3) The ability to modify a program so that recertification is possible.[1]

The goal is to organize complexities, master multitude, and avoid its bastard chaos as effectively as possible.[2]

However, unlike many types of engineers, the software engineer has had few tools, either for implementation or analysis, with which to accomplish his task.

Many of the problems in operating systems which occurred during the mid-sixties can be traced to an inability to enforce the design disciplines indicated by good engineering practices, or to determine after the fact that they had been applied. In some cases the faults appeared years after the system was in the field. Higher level implementation languages for software remove many trivial coding errors and deal effectively with the problem of storage allocation; however, they do little in reducing the major problem of complexity—inter-module communication, software/hardware interface conflicts, and mishandling of real and apparent concurrency within the hardware/software system. This is more obvious when one remembers that most large design efforts are multiperson, and that software modules and hardware designed by many people must communicate properly at the many interfaces.

The hardware designer is somewhat better off since he can call upon switching algebras, flow table analysis[3] and register transfer languages[4,5] to aid him in the design. Unfortunately, these tools are not amenable to the design and analysis of very large systems, and the designer soon learns to modularize his system and to apply his techniques to several modules of manageable size. It is at the interfaces of these modules though, that problems equivalent to those in software arise.

The net result of this inability to systematically deal large scale complexities has been the late delivery of expensive and buggy computer systems. This is not to suggest that the several successful structural approaches to systematic operating system design[6,7] are insignificant, but rather that the difficulty of enforcing their requisite structural and communication disciplines becomes very great as the size of the target system increases.

Hardware engineers encountered this problem of complexity very early in terms of implementation, and responded by developing computer-aided design (CAD) systems for logic diagram production, package placement, wire routing and mask generation, and simulation and test generation.[8] Many other engineering disciplines have also turned to the computer to help deal with the complexities of large systems.[9] It is ironic, however, that the computer, which has great analytic capability, doesn't often forget details, and can enforce structural and communications disciplines by syntactical analysis, has not, to date, been applied to the conceptual and detailed design of computer systems.

* This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Department of the Army under Contract No. AMSEL-PP-Cn122(CAE).

Project LOGOS was begun in 1968 at Case Western Reserve University to exploit these capabilities by creating a computer-aided design environment for both the hardware and software of large-scale computer systems. An integrated hardware/software design system was chosen because mismatches at that particular interface in a computer system are the most costly and time-consuming to correct. The goals of LOGOS can be simply stated: the creation of a multi-designer environment in which computer system designers can define a system in which a high degree of parallelism or concurrency exists, verify its logical and functional consistency, evaluate its expected performance before implementation, and finally implement the hardware and generate the code for the software. Inherent in any CAD system is a philosophy of target systems structure and an associated representation system which both embodies that philosophy and has a well-defined syntax and semantics. It would be helpful here to briefly describe both to set the stage for a discussion of LOGOS' contributions to the software engineer.

A LAYERED VIEW OF SYSTEM STRUCTURE

From a user's viewpoint, a computer system presents an environment to each user which is characterized by a collection of service facilities.[10] Each facility may be activated and directed according to a well-defined communications discipline. Since users do not, in general, act in coordination, the system facilities must cope with multiple and asynchronous requests for services.

Response to a request activates the facility, an instance of which we shall call a task, and the method of handling multiple requests depends upon the nature of the facility. A single-user facility would queue all requests in excess of one, while a limited resource facility such as a magnetic tape controller with six tape drives would allow six concurrent activations before queueing requests. On the other hand, a fully reentrant software procedure would have no limit to its activations although exhaustion of some other resource such as core memory would impose a limit externally.

Users of facilities very often do not care about how a facility is implemented internally, but rather how it interacts with its environment. This concern with the input/output function of a facility is the "external" or "primitive" view. Conversely, a user and, in particular, a designer may need to know the details of implementation as well as the I/O function of a facility. This is the "internal" view.

A facilities approach to viewing systems immediately gives rise to a hierarchical structure. Many facilities in a system provide essentially identical services, or equivalently, have identical subtasks. These subtasks could be viewed as instances of activation of separate facilities shared among those requiring the particular services. The most primitive shared subtasks in a software system are the machine instructions. By the same token, however, the reading of a text file appears primitive to a compiler using the file system facility, although an internal view of the file system shows that the read file operation is quite complex and uses other shared facilities such as the disk channel.

It is natural, therefore, to structure a system as a partially ordered hierarchy of layers, the highest layer being the interface with the system users, and the lowest, the system primitives. A system primitive for a software system might be a machine language instruction or a library subroutine, while a hardware primitive might be a NAND gate or a four-bit MSI adder. A facility on a lower layer may be activated by a task of a facility existing on a higher layer. Its tasks may, in turn, activate facilities on still lower layers.

Between any two layers, there is an interface partitioning the system into facilities below the interface and users of those facilities above it. Users above present an environment of service requests and arrival rates, while facilities below present an environment of service available and service times.

The ability to "collapse" or look at a facility as a primitive suggests that consistency analysis of a facility could be done by exposing the internal structures, analyzing it, collapsing it, and then analyzing its interactions with its environment as a primitive. This is the only practical way of analyzing large systems, and the representation which accompanies this philosophy allows just that.

Implied in all of this is the existence of an interfacility communications discipline for both data and control. Several might be defined such as Dijkstra's P-V discipline[11] or Multics' mailbox scheme.[12] What is important is that whichever one is chosen, it must be enforced, or the layered model will break down, and the analytic capability afforded by this scheme will be lost.

A facility in general consists of four elements:

(i) Resources of one or more types which may be required by the facility subtasks.

(ii) An enclosing control which determines, based upon resource availability, if a subtask should be activated or if the request should be queued or dismissed.

(iii) A set of algorithms defining the subtasks. Algorithms are called activities; instances of their activation are called processes.

(iv) An interpreter which accepts user directives and determines appropriate action.

A facility need not have all of these elements. A wholly software facility would not have local resources, while a storage allocator would control a resource but have no set of algorithms to be selected by a user.

This philosophy of system structure can be applied to both hardware and software. It is consistent with the structural approach to proving program correctness[13,2,14,15] which is to force the structure of the program text (or representation) to correlate strongly with the structure of the actual computation, thus allowing analysis of the computation by analyzing the structure of the representation by stepwise decomposition.

THE LOGOS REPRESENTATION SYSTEM

The central part of any CAD system is its *representation system* which consists of the design data base in which the description of the target system is accrued, the external representation of this design information, and the translators between the external and internal form. The representation system must satisfy several global constraints.

First, the representation must be sufficiently general to describe all interesting and desirable objects in the set of design objects, while at the same time, it must be sufficiently specific to allow algorithmic consistency and performance analysis. Second, the internal representation must be decomposable into elements which may be implemented directly. Finally the designer must be comfortable with the external representation and the constraints it places upon his freedom of expression.

In the case of LOGOS, the target objects are facilities, which can be described by a number of algorithms implemented in either hardware, software, or a combination of both. Therefore, the representation must be suitable for describing both and must yield the target system implementation directly.

The representation must be consistent with the hierarchical, layered view of system structure. It must, therefore, be declarative in that it must express both the structure and function of the target facility to allow algorithmic consistency and performance analysis. It must allow the design to be described in multiple levels of abstraction to accommodate the primitive and internal views of facilities required for stepwise analysis. This feature is especially important to designers since they tend to work "around" in a design rather than in a strictly top-down or bottom-up manner.

Finally, since many of today's computer systems and those proposed for the next generation contain parallel processing capabilities, the representation must allow the specification of parallelism or concurrency in a natural way and be capable of analyzing its effect on consistency and performance.

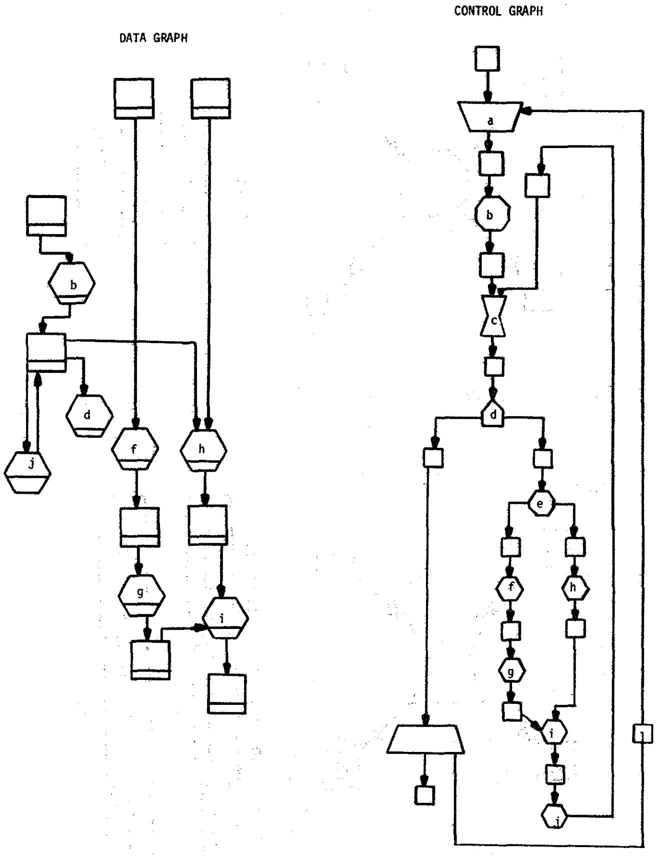
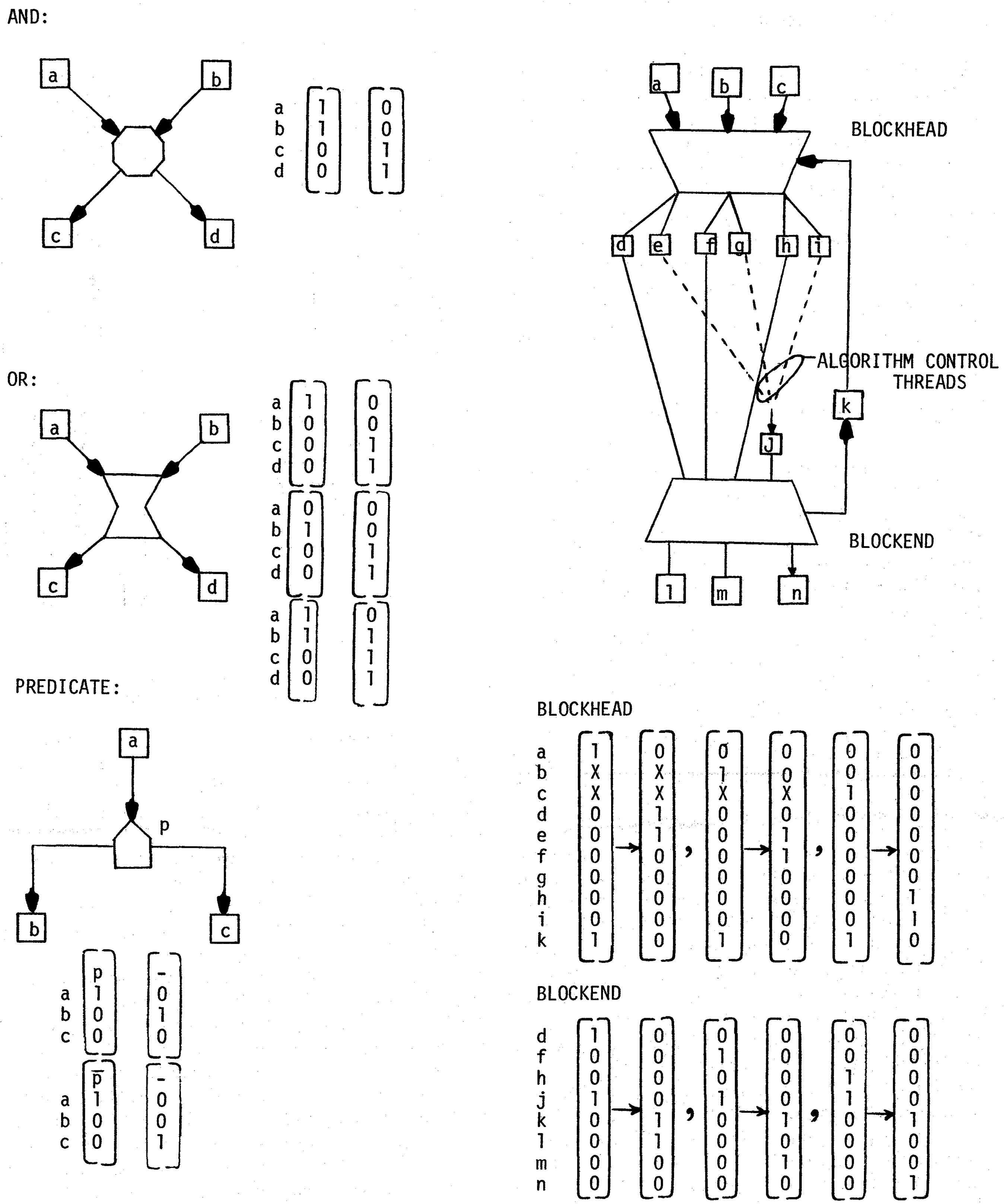


Figure 1—Example of an activity

LOGOS chose a graph-theoretic system of representation which satisfies the above constraints. The system is a synthesis and extension of valuable work done by Petri,[16] Karp and Miller,[17] Holt,[18] Luconi,[19] and others. The extensions were required because (1) LOGOS deals with very large systems and must localize analyses, (2) little work had been done in the representation and analysis of data structures in graph models, and (3) because LOGOS must actually implement rather than merely analyze the target algorithms or systems.

A complete treatment of the representation may be found in References 20 and 21. Briefly, the representation of an algorithm consists of a pair of directed graphs. The data graph (DG) defines the algorithm data structures and the transformations upon them, while the associated control graph (CG) sequences the transformations and defines the control flow. The schema formed by a CG-DG pair is called an activity and will be seen



AND:

$$\begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

OR:

$$\begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

$$\begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

$$\begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

PREDICATE:

$$\begin{array}{c} \\ a \\ b \\ c \end{array} \begin{bmatrix} p \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} - \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{array}{c} \\ a \\ b \\ c \end{array} \begin{bmatrix} \overline{p} \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} - \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

BLOCKHEAD

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ k \end{array} \begin{bmatrix} 1 \\ X \\ X \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ X \\ X \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} , \begin{bmatrix} 0 \\ 1 \\ X \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ X \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} , \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

BLOCKEND

$$\begin{array}{c} d \\ f \\ h \\ j \\ k \\ l \\ m \\ n \end{array} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} , \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} , \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Figure 2—LOGOS atomic control operators

to be the static template of a task. Figure 1 is an example of an activity.

The CG consists of two node types: the squares are control variables or c-cells, and the remaining nodes are control operators. Cells must be connected only to operators by directed arcs and vice versa. There are several types of control operators as denoted by the different shapes in Figure 1. Each type of control operator has an associated enabling or transfer function defined on its input and output c-cells.

The DG consists of cells (squares) which represent the information structures of the activity, and data operators which perform the transformation upon them (e.g., Add, Move, Integrate, etc.). Each data operator is associated with a unique control operator which determines when the data transformation may take place. The initiation of a control operator initiates the associated data operator which then reads its input cells (data structures), performs the data function, and writes the results into its output cells. Upon writing, the data operator communicates to the control operator that it has terminated, and the control operator terminates by alterings its c-cells appropriately.

The flow of control in the CG is determined by the values in the c-cells and the nature of the c-operators to which they are connected. The c-operators are defined so that asynchronous or synchronous control and data flow can be represented. The atomic or first level control operators are shown in Figure 2 together with their transfer functions in vector form.

The AND operator of Figure 2 is used to resynchronize parallel control paths and functions analogously to an AND gate in hardware. The OR operator is asymmetric in that if both of its input c-cells contain 1's, the initiation of the operator preserves the 1 in the second c-cell. It will then reinitiate as soon as possible. This "conservation" of 1's is required to insure determinacy, a property of consistent systems with concurrency which will be discussed later. The OR operator is analogous to an OR gate in all other ways.

The PRED operator is the interface between data values and the control flow in the CG. It is a data dependent control branch whose associated data operator performs a test on its input d-cells. The result of this test conditions the branch in the control.

The blockhead (BH) and Blockend (BE) operators are paired to delineate an activity and form the enclosing control for the facility task being represented. The control algorithms must perform the following functions:

(i) arbitrate access to the facility
(ii) provide a communication discipline between the facility and its users
(iii) define the number of concurrent users which may be served by the facility.

The BH/BE pair described in Figure 2 act as a P-V pair. The arbitration algorithm shown is a fixed left-to-right priority, but round-robin and other disciplines have been implemented also. The BH and BE communicate primarily via the feedback c-cell, which initially contains, if it is present, the number of concurrent activations possible. All control flow is restricted to enter and leave the activity via the BH/BE pair with the exception of nested subroutines or procedure calls (i.e., calls upon activities on the same layer of the system) which are controlled by Call/Return operator pairs constructed from a common control primitive.

These control operators may all be constructed from a common primitive control operator whose definition is logically complete. This primitive operator may be realized directly in hardware, but for the purposes of the software engineer, it is sufficient to state that other, higher-level control operators may be constructed from the primitives and placed in a macrolibrary.

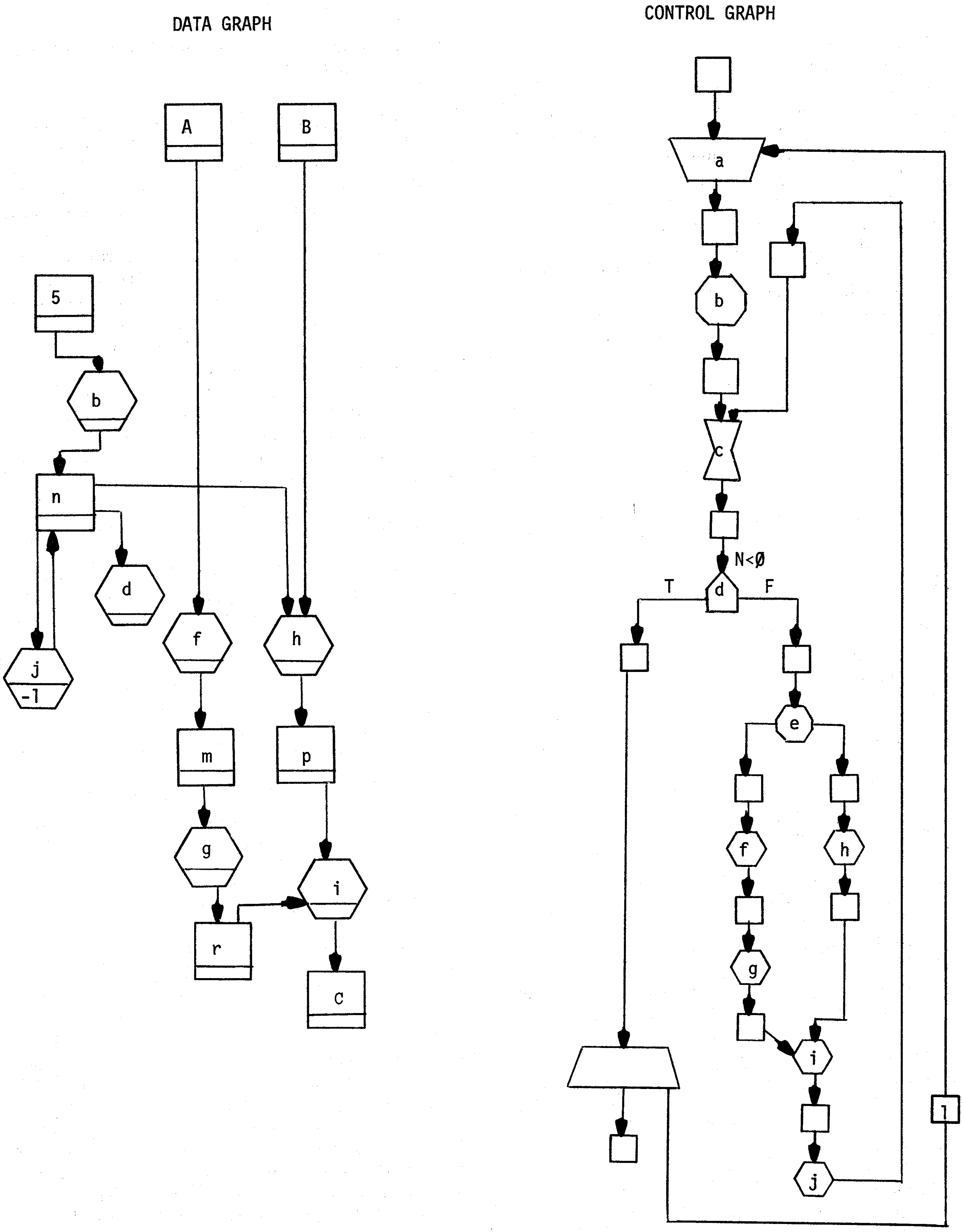
The activity of Figure 1 is shown in Figure 3 with interpretations placed upon the data structures and data operators of the DG (these are informal interpretations; a formal syntax will be introduced later). The activity is the representation of an ALGOL 60 FOR statement with a parallel DO ⟨statement⟩ part. When the task is activated, the stepping variable is initialized to 5, and the loop head is passed. Note that there is no data operator associated with the loophead OR. If the predicate is false, the parallel DO ⟨statement⟩ is executed which allows the sequence of data functions ϵf to be time independent of h. The threads are resynchronized at i whose data operator uses common results. n is decremented and the loop is re-entered. Thus we have represented:

FOR N=5 Step−1 until ∅ DO ⟨statement⟩

The 1 in the feedback c-cell indicates that the activity may be initiated only once before terminating.

The nesting of activities on a layer allows the imposition of an ALGOL-like block structure upon the representation. If the activity in Figure 3 were in a block structured environment, data cells A, B, and C might be global to the block while n, m, p, 5, and r are local.

Thus, a collapsed or primitive view of the activity is that of a single control-data operator pair as shown in Figure 4. For most types of analysis performed by LOGOS, the local structure of an activity is analyzed in its internal or expanded form, and the activity is then collapsed. All further interactions with its global environment are analyzed in the collapsed form. In this



DATA GRAPH

CONTROL GRAPH

Figure 3—Example of Algol 60 for statement representation

way the analysis of a software (or hardware) system can be carried out in a stepwise, computationally efficient manner.

The imposition of an ALGOL environment is optional, of course, and does not affect the representation itself. To do so does imply the existence of an ALGOL-like run time environment layer which implements the necessary storage allocation and other semantics. A cactus stack is required to keep track of the concurrently active and executing tasks.

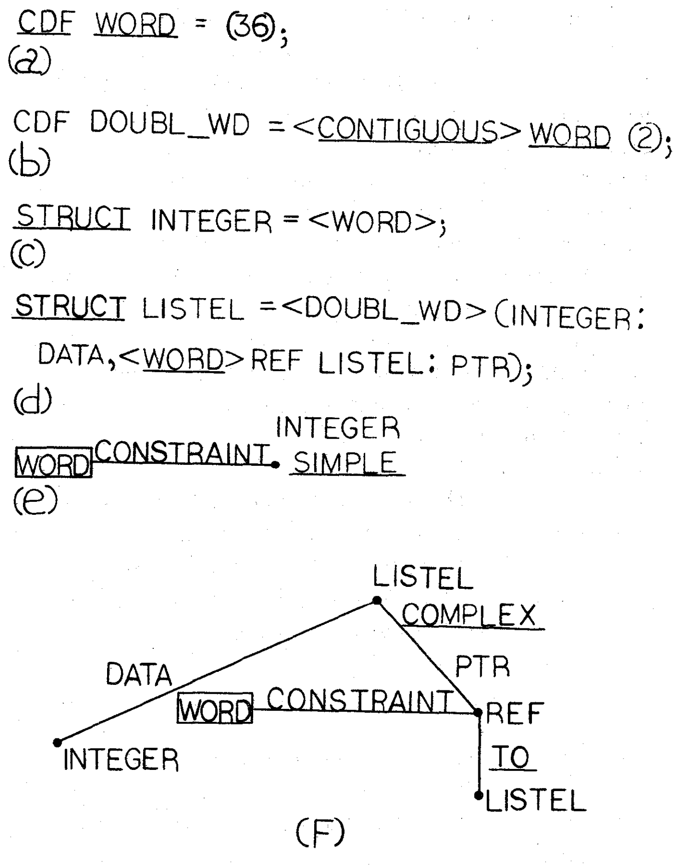
The representation may be generalized to allow control variable contents to be non-negative integers with the control operator definitions changed to allow decrementing of input c-cells and incrementing of output c-cells by greater than one. The constraint that output cells be Ø before initiation of the control operator is removed, and the initiation and termination of control and data operators are made distinct to allow multiple initiations of data operators before termination of preceding activations as resources allow. This generalization is useful in describing higher level software processes and hardware such as pipeline systems.

Thus far, only an ALGOL-type level structure has been suggested. Where does layering enter the picture? The concept of layers enters the representation at the data operator. The function performed by a data operator may be truly a system primitive or it may be a "call" on a lower layer facility. That is, its data cells may be parameters to a task existing on a lower layer which is activated by the initiation of the data operator. This may in turn activate other tasks on still lower layers, but the entire data function appears primitive at the layer on which it is initiated. This is an explicit call upon a lower layer. An implicit call would be the activation of the storage allocator upon activation of an activity in a block structured environment.

A formal syntax and semantics for data structures and a syntax for data operator declarations is being developed. The declarative language is similar to ALGOL 68 and the resulting graphic representation of the data structure descriptors resembles those of Early's VERS.[22] The language consists of six basic building block structures—SIMPLE, MULTIPLE BIT STRUCTURE (MBS), ARRAY, REFERENCE, UNION, and COMPLEX. Examples of SIMPLE structures are integers, reals, etc. MBS's are used to define fields in words or tables. REFERENCE structures denote address variables. UNION is meant in the Set sense, and thus UNION is a place holder for one of a set of structure types. A COMPLEX structure is heterogeneous, consisting of more than one type of basic building block.

Another fundamental concept is that of a constraint. The data structure declarations define logical relationships only. Constraints are used to relate these to physical realities such as words, right half words, etc. These two primitive constraints are: *WORD* and *CONTIGUOUS*.

As an example, consider Figure 5. The length of a WORD is defined in Figure 5a. The constraint



```
CDF WORD = (36);
(a)

CDF DOUBL_WD = <CONTIGUOUS> WORD (2);
(b)

STRUCT INTEGER = <WORD>;
(c)

STRUCT LISTEL = <DOUBL_WD> (INTEGER:
  DATA, <WORD> REF LISTEL: PTR);
(d)
```

Figure 5—Example of data structure declaration

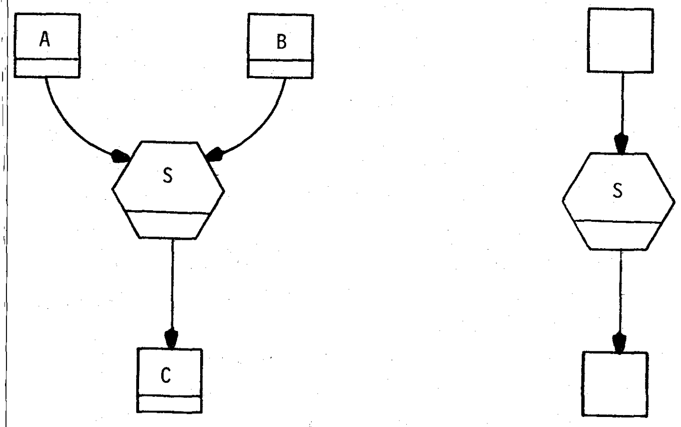


Figure 4—Collapsed activity

DOUBL_WD is defined in Figure 5b, and an integer is Figure 5c. A complex structure LISTEL (list element) is defined in Figure 5d. It is constrained to occupy a double word, one being an integer, and the other a reference to a LISTEL. The terms DATA and PTR are accessing function names. Figures 5e, f, and g show the graphic representation of the resulting templates. Instances of these data structures may be declared which create descriptors based upon the template but with nodes for allocation information added.

Data operators are defined in terms of the types of their input and output data structures. LOGOS has no formal semantics for data operators, so functional definition is not possible at present. However, an informal semantics is being developed to enhance inter-designer communication and to allow simulation of activities if desired.

The intent of this brief and incomplete description of the structural philosophy and representation system of LOGOS has been to set the stage for a discussion of the use of LOGOS and the analysis tools it provides the software engineer and systems designer.

THE DESIGNER'S ENVIRONMENT

Before discussing the types of analysis tools LOGOS provides the software engineer and system designer it would be helpful to examine the LOGOS environment by describing a typical design scenario.

The systems architects, two or three highly skilled analysts, will either be given or will create a specification for the target system in terms of capabilities, number of users, service times, arrival rates, etc. They will pick the design parameters, block the system into facilities, and identify any obviously common facilities such as memory. In the case of a software system built on an existing computer, the system primitives—the machine instructions—will be specified in advance. The facilities will probably be specified in terms of their external characteristics and will have required performance parameters associated with them.

The information will be given to a group of designers (perhaps the architects themselves) who will define these facilities in the LOGOS design data base from their graphics consoles. The individual tasks performed by facilities will be roughed in, and performance parameters defined for them from those on the facility itself.

The designers may define canonical schemata and store them in a macrolibrary to be inserted and expanded during the design process. These may include structures such as IFTHENELSE and DOWHILE, the primary control elements of structured programming.[23] In terms of hardware, these macros will include the set of MSI functions available to the designers.

As the description of a task becomes complete on a layer, the resulting activities can be analyzed, and the activity collapsed. Common tasks may be grouped into facilities on lower layers and defined accordingly, each having a performance specification derived from above. The designers will make their work available to each other by placing it in a common global data base. Here, lower layer facilities common to several designers may be identified. Duplicate and similar facilities and tasks will be replaced by commonly shared facilities.

Modifications may be evaluated along the way by substituting modified tasks into the data base and reanalyzing the affected portion of the design. This process is continued across descending layers until the data operator functions are in terms of the software primitives of the target system, i.e., machine instructions, implementation language statements, or a trial set of instructions if the entire hardware/software system is being created. Code optimization can then be attempted using one of the newer graph-oriented optimization techniques. Code generation will be discussed briefly in the next section.

If the hardware and/or implementation language exists, actual times will be available for the software primitive operations. These can be reflected up and across layers to determine if the performance requirements were met. If not, redefinition of tasks and/or layer boundaries will be required to attempt to meet the specifications.

If the hardware has not yet been designed, it can be begun at this point with the trial instructions and their required times as the target. The process is identical to the one above, but the lowest layer hardware primitives will be hardware elements such as NAND gates, MSI chips, etc. Once again, actual performance information becomes available and is backed out to the software layers.

If the resulting performance is inadequate, the interpreter (hardware) can be speeded up by increasing the degree of parallelism or upgrading the technology. On the other hand, the hardware/software interface can be adjusted by redefining as primitives certain key data operators which were originally implemented as calls upon lower layer facilities. These procedures may be applied interactively in combination to reach the desired performance, if indeed, it is attainable at all. Once the instruction set is frozen, code may be generated, and the necessary steps taken for implementation of the hardware.

Note that this series of events is a departure from the usual practice of defining the target instruction set as almost the first step in system design and then sending the hardware designers away to build a computer and the programmers to build an operating system. The integrated design approach advocated here should (1) reduce the hardware/software interface mismatches, and (2) allow cost/performance tradeoffs to be intelligently evaluated at the proper time—before commitment to hardware and code.

The data structures, data operators and resulting code of the operating system are simply data in one of the data structures—memory—of the interpreter (hardware processor). This is true of all program/interpreter systems. If the interpreter were not to be implemented in hardware but on, say, an 1108, then the data operator primitives would be 1108 machine instructions, and code rather than hardware would be generated.

In addition to a framework for representing layered systems, LOGOS will provide the designer with several types of consistency and performance analyses. Further, code generation of target system software, and ultimate implementation of target system hardware are goals which appear attainable.

The analyses can be separated into two classes—uninterpreted and interpreted. Uninterpreted analysis implies that no interpretation is placed upon the function performed by the data operators for purposes of the analysis. Thus, uninterpreted analyses deal primarily with the control graphs and are topological in nature.

The addition of parallelism or concurrency to an activity raises several analysis questions. Of primary interest is whether multiple activations of a parallel activity (schema) with a given initial control state (contents of its c-cells) and data values will result in the same final values in a set of "result" locations. This condition is called determinacy and was formulated originally by Karp and Miller.[17] This condition, even after formalization, is mathematically difficult to prove. Another condition, more stringent but easier to verify, has been formulated by Karp and Miller.

1. A schema is determinate if, given an initial state, q_o and an initial set of values, each data location has a fixed sequence of values.

With this condition satisfied, then a schema will surely produce consistent values in the "result" locations provided that the algorithm terminates. Karp and Miller further showed that the above condition is equivalent to the following two conditions.

2. (i) No two data operations can be concurrently enabled to "write" into a common data location. (ii) No data operation can be enabled to "write" into a data location while another data operation is simultaneously enabled to "read" from the same location.

From conditions (i) and (ii), a schema is determinate provided that it is free of "race" conditions of two types. This situation should not startle hardware designers who have always faced this problem.

Karp and Miller gave conditions on a parallel schema which allow determinacy analysis to be conducted on the control graph. The analysis tool is a mathematical construct called a "vector addition system" (VAS); for a given schema, the vectors used have one component corresponding to each c-cell in the control graph. A vector q_o gives the initial control state, and, for each control operator, e, a vector δ_e gives control state changes when control operator e occurs. These change vectors may be derived from those shown in Figure 2, but may be generalized to integers greater than ± 1 for higher level representation. A "tree of nodes" is generated from the root node q_o which corresponds to the tree of attainable control states of the schema. The algorithm identifies loops in the control and may be used for finite and infinite attainable state schemata. A complete treatment may be found in Reference 20.

The resulting tree can be used to determine those control operators which can be simultaneously enabled, and, hence, those data operators which are concurrent.

By examining the input and output data cells of those data operators, conditions (i) and (ii) above can be verified. The blockhead/blockend of the activity in LOGOS limit the scope of the analysis, and thus can limit the size of the tree to manageable size. The activity can be analyzed for determinacy and collapsed. It will then appear as a single operator pair in more global analyses.

The vector addition system can be used to check for proper termination of an activity, i.e., can a control/data operator pair remain enabled after the blockend of an activity is enabled? Further, is the topology of the control graph such that the activity will not terminate? Remember that this is uninterpreted analysis, and, consequently, the results of predicate operations are not known. Therefore, in some cases, all that can be said is that there exists a path which if taken, will result in no termination.

Similarly, by viewing all activities as primitives, a potential recursion analysis can be carried out using the vector addition system. These types of analyses fall into the category of general control path analysis, and additional algorithms in this family can be identified and easily implemented using the VAS.

A major weakness in the integrity of computer systems has been the management of system resources and the prevention of system deadlocks. This is particularly true in systems with a high degree of real or apparent concurrency. This problem has been extensively studied, and much insight has been gained.[6,11,24,25] Holt[25] has developed graph models for deadlock and resource allocation which are directly applicable to the LOGOS environment. Resources are represented as control cells, and a topological analysis using adaptations of Holt's results can be performed. Once again, a layered structure tends to limit the scope of analysis.

System performance analysis depends upon knowledge of arrival rate and service request distributions, and, thus is only as good as the model load. However, actual path transit times can be computed in the LOGOS environment, and if model service request distributions and arrival rates are available, performance statistics can be gathered before implementation using a combination of path analysis and simulation, if necessary.

Interpreted analysis deals with the correctness of the algorithms used in implementing the activities. At present, LOGOS has no automated solution to the program correctness problem. The layered structure of target systems, together with the communications disciplines enforced by the syntax of the representation and the various other analysis algorithms tend to assure logical and structural consistency. However, a logically consistent, *but* incorrect algorithm is undetectable. Current work by Scott and Strachey,[26] leading toward a formal mathematical theory of hierarchical systems and semantics, may well be the answer. Results of this work could be adapted to replace LOGOS current data graph syntax and semantics and provide a certifiable representation. In the interim, interpreted data analysis algorithms based upon the functional attributes of the data operators are being considered. For example, a data operator must access data structures of the appropriate type and compute results which correspond to the types of output data structures to which it is connected. This is useful in analyzing data functions which are implemented by interlayer facility activations. In critical areas, actual simulation of the algorithms in question may be performed directly.

Finally, if a global semantic such as ALGOL 60 or FORTRAN is imposed, environmental consistency algorithms such as scope of reference can be included modularly.

CURRENT STATE OF LOGOS

The LOGOS system is being implemented on a Digital Equipment PDP-10 with a Bolt, Beranek and Newman paging box and TENEX executive system. The primary graphics terminals are two IMLAC PDS-1 display systems which communicate with the PDP-10 at 9600 baud. The implementation language is SAIL (Stanford Artificial Intelligence Language). A multi-designer data base management system is being implemented using the LEAP associative data structures of SAIL and the TENEX virtual memory facilities. The system provides for local (single user) and global (shared) data bases with linking between local and global information in a controlled manner. The data base management system is based upon earlier work by M. Pliner.[27]

The graphical representation system is implemented together with the following analysis algorithms: graphical syntax checking, determinacy, halting and termination, and repetition freeness. Implementation of generalized control path analysis is also under way.

The remainder of the control analyses, deadlock and resource allocation, are scheduled to be implemented and integrated by September 1973. It should be noted here that all of the analysis packages are modular and act upon the standard internal representation, thus allowing new packages to be added when necessary.

The implementation specifications for the data structures and data operators are scheduled for completion in December 1972, and implementations should be complete by September 1973 along with the associated analysis routines. These analysis routines assume a FORTRAN environment with a static block structure but may be replaced if another semantic is chosen.

Performance analysis algorithms should be implemented and integrated by September 1973.

Thus, with the very major exception of a formal semantics and corresponding attack on program correctness, LOGOS is scheduled to have a running representation and analysis system by September 1973.

The implementation of target systems requires the production of a code generator for the software and a "hardware compiler" for the hardware portions of the representation. Here again, Scott's work may provide a general solution to the semantics problem for the code generators, but even without such results, if the software primitives in the data graphs are machine language instructions of the target machine, code generation becomes rather straightforward. In addition, the graphic form of the program tasks will allow application of the newer optimization techniques to the target software. A first cut code generation scheme for sequential (rather than parallel) systems should be implemented in early 1974.

Rather than re-create a "hardware compiler" which would require 30-50 man years, LOGOS has chosen to interface with existing hardware CAD systems at the logic equation/logic diagram level. Although much of the information which could help in optimization of the hardware will be lost in going to the equations, the time scale and scope of the project preclude attacking the hardware CAD problem directly. It is felt, however, that the graphic representation may provide helpful insight in the partitioning and placement operations of hardware CAD, and those problems will continue to be studied. The hardware equation/diagram outputs are scheduled for September 1974.

In parallel with these efforts, an attempt is being made to define one or more programming languages to serve as alternate external representations of the target system rather than the current graphical representation. This is being done because some programmers may feel uncomfortable with the graph form, and because the human engineering and scope management problems become significant as the complexity of the target graphs increases.

The LOGOS representation has been used off-line to describe various types of small systems and subsystems including a PDP-8. The resulting descriptions are concise, and being able to see both the structure and function of the systems in one "picture" aids in understanding the target system.

With regard to implementation, the resident executive in the IMLAC display processors was designed according to the LOGOS structural philosophy.

The IMLAC system provides the designer with facilities of (1) creating a picture and designating it a subroutine for transmission to the PDP-10, (2) editing a subroutine, and (3) deleting a subroutine. The system exists on six layers as shown in Figure 6. The lowest is the PDS-1 hardware used by all higher layers. The next layer facility is the character transmitter (all messages, text and graphics are sent to the PDP-10 as multiple character strings). Layer 4 contains the keyboard character handler and the character receiver both of which are users of the character transmitting facility. The character receiver uses the character transmitter facility to control the transmission of characters from the PDP-10 to the IMLAC. The next layer has three independent facilities—the light pen tracking facilities, the graphics message handler, and the core management facility. All of these facilities are used by the facilities on layers 1 and 2, the subroutine edit and subroutine create and delete facilities. The communications discipline between the facilities are well-disciplined according to LOGOS design principles.

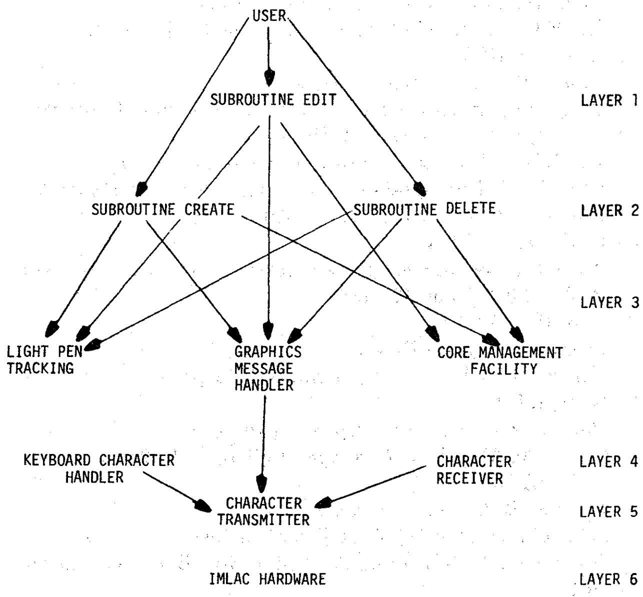


Figure 6—Layer structure of Imlac executive

The design and implementation of the executive required about six man months of effort. It occupies approximately 3000 words in IMLAC core and was coded in assembly language. As with the 'THE'[6] and 'VENUS'[7] systems, coding errors were discovered, but few logical errors were committed in the design. These proved easy to identify and correct.

CONCLUSION

The aim of Project LOGOS is to provide the computer system designer with a computer-assisted design environment in which good engineering practice can be applied to large-scale target systems and verified after the fact. The basis of this good engineering practice is a structural view of computer systems which is a generalization of Dijkstra's[2] and Mills'[23] structured programming for sequential software. Dijkstra's 'THE' system[6] is a result of this philosophy as is the VENUS system.[7] Both these and the IMLAC executive have demonstrated the payoffs of a well-disciplined approach to structure. They were implemented in a fairly short time by small design groups (VENUS required about 6 man years for the design and implementation of the operating system and the support software). They were easily checked out and modified, and have proven to be stable, reliable systems. The primary contribution of LOGOS in this area is that it provides a uniform, analyzable representation in which to express these otherwise abstract notions of system structure, one which leads directly to the implementation of the target software or hardware. It also allows the designer to express the maximum degree of real and apparent concurrency in his target system and provides the analyses required to evaluate its effect.

Both 'THE' and VENUS are small operating systems implemented on small to medium scale machines, yet even they were found to contain a few errors resulting from breaches of discipline. True, these errors were easily corrected, but as the size and complexity of the operating system and hardware increases, the difficulties of enforcing the disciplines, detecting errors, and correcting them without introducing more will increase nonlinearly. It is because of this complexity explosion that a CAD environment such as LOGOS is required for large scale systems.

A LOGOS-type system can provide several other advantages to the software engineer and system designer. First, because performance measurements can be made before rather than after implementation, modifications to the system can be proposed and their effects evaluated economically. In particular, the final positioning of the hardware software interface can be postponed until quite late in the design cycle and can be made a true function of performance vs. cost.

Second, the design team will tend to be smaller. The computer will act as the "bookkeeper" and will perform many of the analyses which have traditionally been attempted manually or not at all.

Third, the increased degree to which a target system can be certified before implementation (even without formal semantics) should reduce the integration and checkout cycle significantly. It may also be possible to produce more complete diagnostics in a LOGOS environment since the entire system description as well as its implementation is stored within the design data base. This is an area for continued research.

Finally, although this hints of "big brother," valuable management and scheduling information can be extracted from such a system. The effectiveness of designers, the times required to complete various portions of the system, etc., could be used in estimating, staffing, and scheduling future systems.

LOGOS is an open-ended system. Although a first producing system will be complete in 1974, it is expected that the users themselves will enhance, modify and tailor the design environment to their needs as new technology becomes available.

ACKNOWLEDGMENTS

The LOGOS design environment described in this paper is the result of work done over the past three years by Professor E. L. Glaser, principal investigator; Dr. F. T. Bradshaw; S. Katzke; the author and several others. In particular, much of the philosophy of system structure which underlies the LOGOS system was articulated by Dr. Bradshaw, and the syntax and semantics of target system data structures and data operators were developed by S. Katzke during his doctoral research.

REFERENCES

1 F G HEATH C W ROSE
The case for integrated hardware/software design with CAD implications
IEEE Computer Conference Digest September 1972

2 E W DIJKSTRA
EWD249—notes on structured programming
T. H. Report 70—Wsk—03
Technological University Eindhoven Netherlands April 1970

3 T BREDT
A model for parallel computer systems
Technical Report No 5 STAN-CS-70-160 Stanford University April 1970

4 C G BELL A NEWELL
Computer-structures: reading and examples
McGraw-Hill Book Company New York New York 1971

5 M BARAY Y H SU
A digital system modelling philosophy and design language
Proceedings Eighth Annual Design Automation Workshop 1971

6 E W DIJKSTRA
The structure of the T.H.E.—multiprogramming system
Comm ACM Vol 11 No 5 May 1968 pp 341-346

7 B LISKOV
The design of the VENUS operating system
Comm ACM Vol 15 No 3 March 1972 pp 144-149

8 C D MARSH
Automation of the design and manufacturing of a large digital computer
IEE Electronics & Power October 1970 pp 375-379

9 M R CORLEY
The graphically accessed interactive design of thermally stressed pipe systems
Proceedings Ninth Annual Design Automation Workshop 1972

10 F T BRADSHAW
Some structural ideas for computer systems
IEEE Computer Conference Digest September 1972

11 E W DIJKSTRA
Co-operating sequential processes
Programming Languages ed F Genuys Academic Press 1968

12 M J SPIER E I ORGANICK
The MULTICS interprocess communication facility
Second ACM Symposium on Operating Systems Principles Princeton University October 1969

13 E W DIJKSTRA
A constructive approach to the problem of program correctness
BIT Vol 8 1968 pp 174-186

14 C A R HOARE
Proof of a program FIND
Comm ACM Vol 14 No 1 January 1971 pp 39-45

15 N WIRTH
Program development by stepwise refinement
Comm ACM Vol 14 No 4 April 1971 pp 221-227

16 C A PETRI
Kommunikation mit automaten
Schriften des Reinsch-West Falischen Inst
Instrumentelle Math und der Universitat Bonn Nr 2 Bonn 1962

17 R M KARP R E MILLER
Parallel program schemata
Journal of Computer and System Sci 3 1969 pp 147-195

18 A W HOLT F COMMONER
Events and conditions an approach to the description and analysis of dynamic systems
Third Semi-annual Technical Report Part II For the Project Research in Machine-Independent Software Programming Applied Data Research Inc April 1970

19 F L LUCONI
Asynchronous computational structures
Doctoral Thesis MIT Cambridge Mass January 1968

20 F T BRADSHAW
Structure and representation of digital computer systems
Jenning Computing Center Report No 1114 Case Western Reserve University Cleveland Ohio January 1971

21 C W ROSE
A system of representation for general purpose digital computer systems
Jennings Computing Center Report No 1113 Case Western Reserve University Cleveland Ohio August 1970

22 J EARLY
Toward an understanding of data structures
Comm ACM Vol 14 No 10 pp 617-627

23 H D MILLS
Mathematical foundations for structured programming
FSC72-6012 Federal Systems Division International Business Machines Corporation Gaithersburg Maryland February 1972

24 A N HABERMANN
Prevention of system deadlocks
Comm ACM Vol 12 No 7 July 1969 pp 373-385

25 R C HOLT
On deadlock in computer systems
Doctoral Dissertation Cornell University Ithaca New York January 1971

26 D SCOTT C STRACHEY
Toward a mathematical semantics for computer languages
Tech Monograph PRG-6 Oxford University Computing Laboratory August 1971

27 M S PLINER
PDMS—a primitive data base management system for representing structured data in an information sharing environment
Doctoral Dissertation Case Western Reserve University Cleveland Ohio September 1971

Some conclusions from an experiment in software engineering techniques

by DAVID L. PARNAS

Carnegie-Mellon University
Pittsburgh, Pennsylvania

In two earlier reports[1,2] we have suggested some techniques to be used producing software with many programmers. The techniques were especially suitable for software which would exist in many versions due to modifications in methods or applications. These techniques have been taught in an undergraduate course[3] and used in an experimental project in that course. The purpose of this report is to describe the results that have been obtained and to discuss some conclusions which we have reached. The experiment was completely uncontrolled, the programmers generally inexperienced and poor, and the programming system used was not designed for the task. The numerical data presented below have no real value. We include them primarily as an illustration of the type of result that can be obtained by use of the techniques described in the earlier reports. We consider these results a drastic improvement over the state of the art. *Major* changes in a system can be confined to *well-defined*, *small*, subsystems. No intellectual effort is required in the final assembly or "integration" phase.

THE PROJECT

The class was asked to produce the KWIC index system described in Reference 2. The project was divided into six modules, but two were combined because they were clearly simpler than the remaining four.* For each of the five assignments we specified four distinct types of implementation. Each student was given one of those to program. Had the experiment been a complete success, any combination of one version of each assignment would have run correctly; we would have had 4^5 working versions (five independent selections from sets of four elements). In addition, each student was assigned to write a program which would "checkout" some module other than his own. Because of the billing policies of our University Computing Center, the programs were to be written and run in WATFIV—a version of FORTRAN. All the defined functions were to be made available as either subprograms or FORTRAN functions.

Only minor additional information was supplied beyond the specifications given in Reference 2.

(1) Where necessary, the error routines were given an additional parameter to be used in identifying the module whose error procedure should be executed. This arose only where the same function could be called from more than one module.
(2) Module identification numbers were assigned for use in selecting the error routine.
(3) Conventions for the naming of labelled common were established. No programmer ever knew the name of the common used by other programmers. The conventions merely avoided duplication.
(4) Maximum values for the various parameters were specified.

The students did not know which combinations of systems would be tested, nor did they know which version of the module they would check. For that reason they could use no information other than the published specifications.

On completion of the programming and checkout of individual modules, complete systems were assembled by a graduate student who had absolutely no knowledge of the internal structure of any module. The results indicated below were obtained with only one major difficulty. All students had dimensioned their arrays for the maximum possible values of the parameters. The combined storage exceeded what was available in the programming system. The sizes of the arrays were easily reduced to a value appropriate for the actual

* See Appendix 1 for a brief description.

TABLE I—Final State of Assignments for Individual Participants

	Assignment				
Version	1	2	3	4	5
A	OK	OK	OK	NOT COMPLETED[2]	OK
B	OK	OK	OK	OK	INCORRECT[4]
C	INCORRECT[3]	STUDENT DROPPED	NOT ASSIGNED	OK	INCORRECT[5]
D	INCORRECT[3]	OK	OK	NOT ASSIGNED	OK
E	NOT ASSIGNED	OK	OK	OK	NOT ASSIGNED

Notes:

1. In our calculation of the potential number of working combinations we excluded versions which were not assigned or were assigned to students who did not complete the course.
2. No work was supplied by this student.
3. The students assigned to check these programs did not do so. The modules were thought (by the instructors) to be incorrect, but the simplest test was to include them in combinations with programs which were working properly. The suspect programs made errors *which were detected by the other modules.* The errors were verified by the instructors to be violations of the specification of the modules in question. In fact, in both cases the error had been detected by the student's own tests, but they failed to examine the output closely enough to notice. (These were, by any measure, two of the poorest students in the class.)
4. This program was clearly incorrect, but still did not violate the restrictions specified for the modules which it called. Thus combinations involving this program would run but would produce incorrect output. It produced the same incorrect output in every combination tested. The program was "completed" by the student well past the due date and the "checker" was not able to do his job.
5. This program simply failed to terminate in any case. The error was found by the checker.

test. (In a language such as ALGOL where the dimensions to arrays could be variables, this difficulty would have been easily avoided.)

Table I gives the versions of each module which we judged correct. From this we may calculate that there are 192 working combinations. We could not test all of these. An experiment was planned so that (1) each version is used in at least two combinations and (2) each version was in at least one combination where it was the only difference with another tested combination. Table II shows the results.

It should be noted that the fact that only 192 of the possible 1024 combinations worked does not represent a failure of the method. It represents the failure of five students out of 20 to complete the work assigned to them. One can argue that these failures provide additional evidence of the value of the method. In each case it was possible to show, without doubt, that the individual student had failed to do his assignment. In most projects to construct programs in teams some ambiguity in the individual work assignments results in some difficulties which cannot be assigned to an individual programmer. Because of the use of formal specifications in this project we had no cases in which a program was found to meet its specifications yet would not work in combination with other programs which met their specifications.

Further experimentation

1. When an earlier version of this note was circulated privately early this year, Mr. Thibault of IRIA, Rocquencourt, France studied the data and suggested trying the combination 1B,2B,3D 4E and 5D which he believed would be significantly faster than any of those tested.[4] It ran in 4.4 seconds.

TABLE II—Execution Times for Some of the Combinations Tested

Combination Tested					Execution Time (sec.) (excludes compilation of 6-8 sec.)
1A	2B	3B	4B	5A	37.26
1A	2D	3D	4B	5A	11.42
1A	2D	3A	4C	5A	10.87
1B	2E	3A	4C	5A	10.31
1A	2E	3A	4B	5D	8.53
1B	2A	3E	4C	5B	21.79
1A	2A	3B	4B	5B	302.99
1A	2A	3B	4B	5A	50.16
1A	2A	3B	4C	5A	36.69
1A	2A	3D	4C	5A	11.07
1A	2B	3D	4C	5A	10.99
1A	2A	3B	4E	5A	43.30
1A	2D	3B	4E	5A	43.61
1A	2D	3B	4E	5D	19.17
1A	2E	3B	4E	5D	19.16
1A	2E	3B	4C	5D	28.48
1A	2B	3B	4C	5D	27.23
1A	2B	3D	4C	5D	8.43
1A	2B	3D	4C	5B	76.34
1A	2B	3D	4B	5B	113.32
1A	2B	3B	4C	5B	238.88
1A	2B	3E	4C	5D	10.06

2. We have just repeated the whole experiment with a somewhat larger class. The results were essentially the same. We estimate that the family of programs has 1100 members, more than 400 of these were tested. Performance improves somewhat, ranging between 3 and 13 seconds. The only interesting distinction between the two experiments was that the instructor (project leader) changed from intensely interested to bored and unconcerned with no noticeable effect. We also eliminated the problem with storage limitations mentioned above.

Conclusions

1. We cannot avoid stating our conclusion that the experiment has revealed some validity in the comments of our earlier papers.[2,3] Clearly one purpose of this paper is to draw your attention to those earlier ones.
2. Our most significant new conclusion comes in the area sometimes called "project management". Recent papers have suggested that the project manager must devote a significant part of its best manpower to the "integration phase". In our experiment the "integration phase," while not mechanized, was so simple that it could have been mechanized. Even in the few cases where errors did occur, the system had been structured in such a way that diagnostic messages automatically indicated the module making the error. We had no need for anyone who had a thorough knowledge of the whole system. Our experience indeed suggests that the integration phase is a very poor place to invest one's manpower. The limited capacity of our minds makes us more efficient when our job depends on a relatively small amount of knowledge. Moreover, if we plan our project management around a large "integration phase," we will have to invest that manpower again whenever we change some part of the system.

   Our experiment suggests that manpower can be much more profitably invested in the "preprogramming" or "design" phase. The success of our project depended largely upon the precisely written module specifications described in Reference 1. The "cost" or intellectual effort required to produce one of these module specifications was comparable to the cost of producing an implementation of the module. Such predesign work therefore appears to many as unjustifiable overhead. When we amortize this cost over the number of versions of the system which are finally built, and consider the savings realized in the final "integration" phase, it appears to us that the overhead is well justified.

   Efforts in the industry to invest heavily in a "pre-design" or "concept" phase have often proven fruitless because the outcome was a set of natural language documents which were so general that they provided almost no decisions to guide the development groups. When this predesign phase produces precise module specifications the payoff is much more significant. Additional amortization of the "pre-design" effort can occur when the modules or their specifications are used (either unchanged or slightly modified) in a later project.
3. Another important conclusion lies in the area of documentation. Several firms have invested heavily in formalized documentation standards intending to make all information easily available to everyone on the project. Our experiment suggests that the effort in these projects can be focussed. Precise documentation of the external characteristics of each module is essential and should be in a standard notation. Our project had minimal documentation about the internals of the one-man assignments. Industrial practice would require more effort in the area than we put into it, but much less effort than is now common. More significant, the specifications produced in the pre-design phase were the only external documentation required throughout the project. These documents were updated several times as errors were discovered, but no additional descriptive material was needed. This is yet another way that the effort invested in the pre-design phase can be amortized.
4. Our experience demonstrated the importance of careful attention to the possibility of errors in the running program during the "pre-programming" phase. Because of our careful attention to the errors in the design phase, errors which did occur when the systems were assembled were quickly traced to their source and meaningful diagnostic information was produced with almost no effort on the programmer's part. A paper reporting what we have learned in this area is in preparation.
5. Our experience has indicated the great value of independent module tests (by persons other than the module author) *before* integration. In an earlier effort of this sort we required each programmer to test his own module before integration. In the two experiments which we discuss here, we required an additional person to

test the module against the formal specifications (another use of our pre-design efforts). Our success rate increased drastically and there were apparently two reasons:

(1) Sloppy programmers do sloppy tests.
(2) The specifications, although precise, can be misinterpreted by human programmers. A misinterpretation by the programmer which resulted in an error in his module often results in a corresponding error in his tests. An independently written test was unlikely to share the same misconceptions.

We are well aware that, as E. W. Dijkstra has put it,[6] "Program testing can be used to show the presence of bugs, but never to show their absence." Showing the presence of bugs however is a very valuable service.

We eagerly await the day that professional programmers habitually produce programs which are written so that they can be carefully *proven* to be error free. In the meantime we suggest that effort invested in independent pre-integration testing is well worthwhile.

Our experience also suggests that both the hierarchical structure which can be found in the system[2] and the abstract nature of the modules themselves greatly ease the building of the "scaffolding" required for independent module tests. To test a given module one need simulate only those modules immediately below it in the system hierarchy. Further, the nature of the modules means that many of them can be directly simulated by arrays for testing purposes.

NON-CONCLUSIONS

The reader of this paper and the references might be led to some conclusions which those closer to the project would not draw. We mention them here to avoid misleading our readers.

1. The KWIC index structure given in Reference 2 is the best known. FALSE: Our experiment showed us a number of faults in the design which we are now trying to remedy.
2. Writing a system in a higher level language such as FORTRAN helps to produce a better structured system. FALSE (or at least not supported by our experiment): We used FORTRAN because of the billing and priority policies of our computation center. Use of the language actively interfered with some of our efforts imposing quite unnecessary restrictions on what we did. This was especially apparent in the area of error handling. The secret of our success seems to lie in the module specifications which were language independent.
3. D. L. Parnas is a good project manager. FALSE! Experience has shown him to be absent minded, inattentive to details, unaware of the passage of time, forgetful, etc., etc. The project succeeded *in spite of* his being in this role.
4. The students in the course were good professional programmers. FALSE! Most of the programs written were horrid by any professional standards. The experiment succeeded in spite of the programmers as well. (There were a few good programs but they were notable exceptions.)
5. Communication between modules should always be by subroutine call as it was in the sample system. FALSE! If one divides a system into modules according to the criteria given in Reference 2 the use of subroutine calls imposes a terrible overhead.

Two more non-conclusions

Several writers (e.g., Dennis[7]) have suggested that a hardware supported virtual memory and a language with the ability to pass complex data structures are necessary conditions for well structured or "modular" programs. Neither of these "necessary conditions" were met in the experimental system we are discussing.

We did not need the ability to pass data structures as parameters (all parameters were integers) between modules because of the nature of the way that our system was divided into modules. Data structures were always operated upon within a single module. We suggest that there is often a false identification of the modular structure seen at design time with characteristics of a program when it is running. This however is a very complex issue and we cannot discuss it further here.

Our programs were written in FORTRAN and could have run either with or without the virtual memory mechanism. This however is begging the question because we built a small system where overlays were not necessary. Memory assignment could be done at compile time or assembly time and would be fixed while the program was running. It is definitely true that memory assignments are data which should not be shared between modules but should be hidden from all but one.[8] This allows (in fact requires) programs to be written for a virtual memory. However, the implementation of the one virtual memory module can be done in many ways (hardware mapping, run time software, or assembly time software.) The choice between these implementations is determined by performance considerations not by "modularity" considertions. Thus we can agree with the virtual memory recommendation only it if is stated more carefully indicating that the necessary condition is that memory allocation considerations be hidden from all but one "module." As a historical note we might mention, that one well-structured system, the T.H.E. operating system (which made heavy use of the virtual memory concept) was implemented without mapping hardware using the run-time software option mentioned earlier.

FINAL CONCLUSIONS

We believe that the small scale experiment described above has provided us with some valuable insights into methods of software production. We recognize the danger of applying small scale results to larger scale projects. We hope however that some organization with the facilities for carrying out larger scale projects will cautiously attempt to apply these results to larger scale projects so that we may refine them further.

REFERENCES

1 D L PARNAS
A technique for software module specification with examples
Communications of the ACM (Programming Techniques Department) May 1972

2 D L PARNAS
On the criteria to be used in decomposing systems into modules
To appear in Communications of the ACM (Programming Techniques Department)

3 D L PARNAS
A course on software engineering
Proceedings of the SIGCSE Second Technical Symposium March 1972

4 M DEPEYROT
Private conversations

5 D SMITH
An organization for successful project management
Proceedings of the 1972 SJCC p 129

6 E W DIJKSTRA
Structure programming
Report on a Conference on Software Engineering Techniques held in Gramish

7 J B DENNIS
Modularity
Course notes from an advanced study institute held at Technical University of Munich February 1972

8 D L PARNAS
Information distribution aspects of design methodology
Proceedings of IFIP Congress 1971 August 1971

APPENDIX I

A Brief Description of the System(s) Built in the Experiment

This appendix is intended for those who have not yet read Reference 2.

The system being built was intended to read in a set of titles and produce an alphabetized listing of all circular shifts of those titles (a KWIC index).

This six modules were:

1. Input—The only module which knew the input format. Programs in this module read the input but called other modules to actually store the data.
2. Output—The only module to know the output format. This program took the information to be printed from other modules, but selected the format of the information of paper.
3. Line-Holder—The only module to know how the titles were stored in memory. The module offered programs which both stored and retrieved the information from memory.
4. Circular-Shifter—The only module to know how the circular shifts were represented in memory. Some versions actually stored all shifts explicitly, others sorted only relatively small directory tables.
5. Symbol Table—This module was hidden within *some* versions of line holder. Programs calling line holder were unaware of the existence of sysmbol table.
6. Alphabetizer—The only module to know the sorting method which was used. Some versions did all sorting initially, others sorted only as needed.

Project SUE as a learning experience

by K. C. SEVCIK, J. W. ATWOOD, M. S. GRUSHCOW,
R. C. HOLT, J. J. HORNING and D. TSICHRITZIS

University of Toronto
Toronto, Ontario, Canada

INTRODUCTION

"It is absurd to separate the study of designing from the practice of design." (Christopher Alexander)

Project SUE at the University of Toronto is developing an operating system for the IBM System/360 family of computers. We are basing our work as much as possible on previous research in operating systems, including that of Dijkstra,[1,2] Lampson,[3] Brinch Hansen,[4] and the MULTICS group.[5–7] Their ideas have been tested in actual systems, but separately, and on uncommon machines. We wish to combine their ideas and the ideas of others in a system for a widely available machine.

In Project SUE, we have attempted not only to build an operating system, but also to learn how to organize a large software project. To this end, we have attempted to structure and document the project itself as well as the system.

We have set high standards for the system. We want it to be efficient. We want it to be extensible in the sense that it is possible to append various protected subsystems which each serve a community of users. Above all, we want the system to be reliable and understandable. We make no attempt to compete with generality of existing operating systems. Rather, we are creating an operating system nucleus (in the sense of Brinch Hansen[4]) which can be extended to support particular applications. The system nucleus is designed to support simultaneously, for example, an interactive system and an independent batch monitor.

This paper presents our objectives, and how they have influenced our selection among reasonable alternatives in some design decisions. This material has not arisen from abstract discussions of the theory of operating systems. It comes from the specification, detailed design and partial implementation of a system. A description of the system design is available elsewhere.[8–9] Here, we will only discuss some aspects of the design process.

SYSTEM STRUCTURE

The concept of *processes* has been useful in understanding and designing operating systems.[2,4,7,10] Each process proceeds asynchronously as if executing on its own (virtual) machine, except when mechanisms for the explicit interaction of processes are invoked. These interactions may depend on relationships among the processes. For example, processes may be

(1) regarded as equals, or
(2) related in a tree-structured hierarchy.

The latter situation arises if each process (after the first one) is created by another process. In what we call the *creation tree*, a process is a *son* of the process which created it, and the *father* of any process which it creates. While equality among processes is a simple relationship, the weak system structure it imposes makes other goals (understanding the system, assuring its reliability, and establishing its correctness) difficult to attain. A hierarchy among the processes specifies a logical order in which to understand them and demonstrate their correctness.

The *virtual machine* upon which a process executes is defined by the set of operations supported for the process by other processes, lower level software, and hardware. Each process should be sufficiently simple that its correct operation can be demonstrated (on the assumption that its virtual machine operates correctly). If the correct operation of a virtual machine does not depend on the correctness of any process using it, then a logical order for understanding and demonstrating the correctness of the system is apparent.

It is possible to distinguish between types of process hierarchies in which virtual machines are

(1) completely ordered,
(2) partially ordered.

In the former case, the system is viewed as a sequence of increasingly sophisticated virtual machines ("onion-like layers"). The lowest level virtual machine corresponds to the hardware, and each higher level is created by adding a layer of software (possibly composed of processes) to the previous virtual machine. Dijkstra has described the T.H.E. system in terms of six levels of virtual machines.[2] Every process in such a system is associated with the level of virtual machine upon which it executes. When virtual machines are completely ordered, any two processes are related in one of two ways; either they have the same virtual machine, or one of them helps provide the virtual machine used by the other. A more general process hierarchy results when virtual machines are only partially ordered. Independent processes then need not have the same virtual machine.

The SUE system structure permits hierarchies of the latter type although several onion-type layers are distinguishable and worthy of mention. The *Kernel* is a layer of software which uses the hardware to implement processes (their creation, destruction and communication), protection, simple management of memory and channels, and timing facilities. The innermost group of processes uses the Kernel to create the *Nucleus*, a more sophisticated virtual machine which provides disk files, peripheral input and output facilities, and mechanisms for measurement and accounting. The virtual machines used by the processes which form the Nucleus can be partially ordered with respect to sophistication. It is possible to create a set of virtual machines which are completely ordered by adding to some of the virtual machines facilities which will not be used by the corresponding processes. This corresponds to imposing an onion-type hierarchy on the processes in the Nucleus.

There are many ways of grouping operating system activities to form processes. At one extreme each conceptually asynchronous activity might be carried out by a separate process. However, asynchronism among activities does not necessarily justify the existence of several processes with frequent interactions among them. For example, at one time we planned to have a process to manage each disk spindle and each disk control unit. However, each disk spindle manager would interact with a control unit manager so frequently that little asynchronous activity would occur. We have concluded that the extra processes are not justified and that all disk spindles and all disk control units should be managed by a single process. Other aggregations of activities have been adopted, and the SUE Nucleus now consists of only seven processes.

COMMUNICATION AND COOPERATION AMONG PROCESSES

Because our operating system is based on the cooperation of processes, the communication mechanisms must be efficient, easy to understand, and easy to use, but secure from unauthorized use and the danger of deadlock.[12,13,14] Many schemes for process interaction have been developed. Each is based on either

(1) shared data, or
(2) message passing.

Process interactions in Dijkstra's T.H.E. system use shared data called semaphores and special indivisible operations for manipulating them.[1] In a message passing scheme suggested by Wulf, each process possesses a number of *ports*, and each port is the interface to a communication link with another process.[15] Establishing the communication link may be an expensive operation, but, with the link established, less checking is necessary as each message is passed. A *mailbox* with several message slots may be inserted in a communication link to provide automatic buffering of messages.[16] Because we desire autonomy among processes in Project SUE, a message passing form of communication is more appropriate than a scheme based on shared data.

Message passing through ports and mailboxes was initially accepted as the mechanism for interprocess communication in Project SUE. Much work was done examining protocols for establishing and using communication links, and making sure that deadlock would not occur.[16] The scheme seemed to be compatible with our goals of making the system efficient and understandable.

Only after much more time and thought did we identify some problems with communication through ports and mailboxes alone. One problem was establishing the communication links. A process must name the process, or class of processes, with which it wishes to communicate. Unless system-wide standard names are established, another significant communication mechanism is required to coordinate the naming of processes.

Processes which provide a service, such as device allocation or file system management have special communication requirements. A large number of processes wish to use each such service. With extensibility as a design goal, the number of processes that can simultaneously have a communication link to a particular service process should not be limited (although such limits exist in many systems). Providing each service process with enough ports to guarantee that competition could not lead to deadlock would require the commitment of an excessive amount of memory. We considered adding a second form of mailbox which could attach a single port of a service process to an unbounded number of other ports. However, the complexity of such objects and new problems in their design compelled us to seek a better solution.

Mailboxes are not appropriate for passing large messages (such as block transfers on input and output). Not only is memory space committed unnecessarily to mailbox buffers, but each message must be moved twice (source to mailbox buffer, then mailbox buffer to destination) when one move could suffice. Finally, more careful analysis indicated that passing even small messages through mailboxes would not be as efficient as we had hoped.

Another mechanism was suggested to supplement communication through mailboxes. We call service processes *facilities* and they are used as are the *monitors* described by Hoare.[17] A requesting process contacts a facility directly by issuing a *facility call.* The call is unbuffered, and the requesting process cannot proceed until the facility completes the requested service.

For the sake of system structure, we restrict which processes are allowed to call on any particular facility. A straightforward mechanism for representing this information is an access matrix, whose (i,j)th element indicates whether process i may call upon facility j. However, such a matrix is not easily kept current in an environment where processes are dynamically created and destroyed. Also the access matrix does not contribute to system structure.

By relating permissibility of facility calls to the creation tree, a hierarchical system structure may be enforced implicitly. Three alternatives we considered are that a facility may be called upon

(1) only by its descendants,
(2) only by its descendants, and by its younger brothers and their descendants, or
(3) only by processes farther from the root of the creation tree.

The second of these represents a compromise between the other two. The first is the simplest and most understandable. The advantage of the second alternative over the first is that processes which provide a facility are not compelled to also create and monitor sons. The third alternative was rejected because the second represents a more structured solution which does not greatly impair flexibility. Choosing between the first and second alternatives was difficult. After several weeks of debate, expediency of implementation caused us to permit facilities to be called upon only by their descendants (first alternative).

The facility call mechanism solves the problems of naming the process to be contacted and of allowing service processes to respond to arbitrarily many customers. We soon realized that facility calls were also sufficient for all other communication needs in the SUE system. Conversations between any two processes are accomplished by facility calls from each to one of their common ancestors. Service requests can be buffered by creating a son to provide the buffering. Thus, in order to reduce the number of different system objects and mechanisms, mailboxes and ports have been eliminated from the SUE System. All interprocess communication is done with the facility call mechanism.

Because all facility calls are directed toward the root of the creation tree, deadlock can be prevented by assuring that each facility completes each user request within finite time. Certain situations require that facility calls be used in an unusual manner. The innermost Nucleus processes are situated in the system structure where disk files and typewriter communication are not available. Yet they need to report error conditions to the operator, and record accounting and measurement information in disk files. Thus the innermost Nucleus processes occasionally require the assistance of a descendant which is at a level of the process hierarchy where disk files and operator communication are available. This descendant, known as the Special Condition Manager, effectively provides service to its ancestors in the creation tree. So that no process waits for a descendant, the facility call mechanism is employed as follows: The Special Condition Manager creates a son for each ancestor which may require its services. Each son issues a facility call on the corresponding ancestor, requesting the next "special condition." When a process wishes service from the Special Condition Manager, it simply completes the service call (which *should* be outstanding) indicating what "special condition" exists. If the son of the Special Condition Manager has not issued the facility call, the process must not wait for the call to occur, but must take some alternative action. Thus, information may be lost if the Special Condition Manager does not react with sufficient speed, but special conditions cannot deadlock the system.

AUTHORIZATION, ACCOUNTING AND MEASUREMENT

An operating system should provide mechanisms to prevent unauthorized and excessive use of system resources. It should also be able to measure resource usage and attribute it to processes or groups of related processes.

The central mechanism in the authorization and accounting functions of the SUE system is the concept of capabilities.[5,18,19] A capability is a control block associated with a process which indicates that the process is authorized to use a particular resource in a particular manner. Processes are not allowed to tamper with the information held in capabilities (especially their own!). The ability to create and modify capabilities is restricted to a carefully protected routine deep in the Kernel.

Capabilities are but one possible way of representing protection information. Lampson has described a theory of protection based on *objects* (resources and processes) and *domains*.[3] A process executes in a particular domain. A domain is defined by the manner in which processes executing in that domain are authorized to use the objects of the system. When processes are units of protection as well as units of asynchronous activity, the domain of a process may be represented as a list of capabilities, one for each resource accessible by the process.

An alternative manner of representing protection information is to associate with each resource a list of processes authorized to use it. This method is less desirable in the SUE system because facilities do not discriminate among processes in providing service. Using the capability representation allows each process to allocate the right to use a facility among itself and its sons.

In the SUE system, we distinguish three varieties of capabilities. One governs resource usage qualitatively (permission to use a disk drive or to access a particular file), while another governs quantitatively (the number of files which may be created or the number of file read operations which may be done). A qualitative capability is known by the Kernel to contain a word of Boolean information representing access rights, while a quantitative capability is known to contain a number. The number may be decremented by the facility whenever the capability is used to request use of resource. This is similar to punching a "meal ticket" each time a meal is consumed. When the count reaches zero, the capability no longer has value. The third variety of capability contains information which is interpreted not by the Kernel, but only by the process which created the capability.

We have chosen not to use two features of general capability schemes. First, we do not allow rights to a resource to be passed between arbitrary processes by transferring a capability. In order to maintain system structure and to avoid difficult questions about how to dispose of leftover capabilities when a process is destroyed, we have restricted all capability transactions to occur between either father and son or facility and user. Second, we do not use capabilities to represent authorization information about some resources needed by every process (such as processor time, and memory space). For resources common to all processes, we use an efficient encoding of authorization information. The routines for manipulating capabilities would be made too complex if they had to deal with each special encoding, so the capability concept is not used for these resources.

During much of the design of the SUE system, we believed that capabilities would have to be associated with longer-lived entities than processes. Consider a permanent disk file which is created by one process, then used from time to time by other processes. The succession of processes each must have a capability for accessing the disk file, yet their life spans need not overlap. We faced the problems of how to keep the capabilities in the system and recognize which capabilities should be given to a particular new process. We planned to have permanent entities, called *sponsors*, whose capabilities would be kept in disk files. Sponsors correspond more or less to people who pay for use of the computer system. Each process which deals with a permanent file will have been initiated on behalf of some person (sponsor). Thus, each disk file capability could be associated with a sponsor, and transferred, upon request, to processes created later on behalf of that sponsor.

Further investigation revealed difficulties with implementing sponsors within the Nucleus. First, it seemed necessary to give the power of transforming data into capabilities to a process whose virtual machine provided disk files, yet we wished to use capabilities to protect disk files. Second, the scheme would increase the complexity of the flow of capabilities in the system. At process creation, capabilities would come not only from the father, but also from the sponsor. Worse yet, at process destruction, a decision mechanism would be needed to determine which of the remaining capabilities were to be returned to the sponsor and which to the father.

We have since found an alternative solution to the problem of capabilities for permanent disk files. Since disk space is to be allocated among the independent sub-systems being supported by the Nucleus, and then subdivided by each among its sons, the responsibility for associating subdivisions of file space with people (or sponsors) can be left to each process which divides its space among its sons. Further, by requiring that each suballocation consist of a subset of the father's file space, the father is able to retain, in a single capability, the authorization to the entire file space which it controls. Only when a subsystem process is destroyed is there still a problem of where to keep the file capability. By requiring the number of independent sub-systems to be bounded and small, we can store the file space capability for each sub-system within the Nucleus. This approach eliminates the problem of transferring capabilities to and from peripheral storage and moves the sponsor structure completely outside the Nucleus. The complexity of a disk resident sponsor structure makes such a move desirable.

Mechanisms for accounting and measurement of resource usage are implemented using capabilities. Essentially, every process is held accountable for the resource usage represented by any capability it receives from its father or a facility. If it does not wish to be financially responsible for the resource usage of its sons, it must record the value of the capabilities passed to each son and the value of the capabilities returned when the son is destroyed. In this manner, resource usage can be attributed to individual processes at as many levels of the system as is desired.

Perhaps the strongest motivation for using capabilities as the mechanism for authorization and resource allocation is that we wish the system to be conveniently extensible. Since we cannot define the universal resource set, we have provided mechanisms so that processes can define arbitrary resources and can authorize and account for their usage.

RELIABILITY AND EXTENSIBILITY

Our original proposal contains the sentence, "A design criterion is that neither the erroneous nor the malicious program shall be able to 'crash' any other user, or the system, under any combination of circumstances." We initially interpreted this as, "No process should ever have to put itself at the mercy of another process." By our selection of system structure and mechanisms for resource allocation and authorization, we have designed a system in which no process can cause incorrect operation of any process which contributes to the support of its virtual machine (that is, any process closer to the Kernel in the creation tree). Further, two processes on different branches of the creation tree cannot interfere with each other. Thus protection in the SUE system provides two-way insulation ("firewalls") between independent, non-interacting processes, and one-way insulation of processes from their descendants in the creation tree. Every process can be mistreated by any facility from which it requests service, and it has no protection against the whims of its father. However, we do not feel that it is a compromise to our original goal to require a process to trust the virtual machine upon which it runs. Rather, we have learned more precisely what our original goal was and how inadequately defined the terms "user" and "system" are.

Because extensibility is among our goals, we are unwilling to establish a clear distinction between "system" processes and "user" processes. Definition of "user" is always *relative* to a particular process. The descendants of any process form the set of potential users of that process. All mechanisms within the Nucleus are intended to be understandable and flexible. They may be helpful to subsystems which are appended to the Nucleus. We hope subsystems will exploit the mechanisms provided within the Nucleus. However, subsystems may choose to conceal the Nucleus mechanisms from their descendants. For example, the distinction among the varieties of capabilities, or, in fact, even the existence of capabilities could be concealed by a subsystem from its users. Similarly, the communication mechanism used within the Nucleus can be replaced. A subsystem might, for example, choose to implement mailboxes as the mechanism for communication among its users.

IMPLEMENTATION LANGUAGE AND PROGRAMMING

Our desire to make the SUE system understandable, modifiable, and extensible along with our desires to facilitate coding and demonstrate correctness made the use of assembly language for implementation unacceptable. Although several systems programming languages have been developed,[20,21,22,23] we chose to use an available compiler generator[24] to design and implement a system language specifically for SUE. This language is documented elsewhere.[8,9,25,26] It features convenient definition of new data types and control structures which facilitate writing understandable programs.

Hoare's first thesis on the use of high level languages in constructing large programs states:[27]

"Programming languages are little help in the construction of large programs.

1. To design a 'language' as part of design and implementation of a big system is essential.
2. To 'implement' this language is disastrous.
3. To use a language designed and implemented for any other special purpose is of doubtful benefit."

Hoare's thesis is a valuable warning of potential danger, but our experience indicates that disaster is not inevitable. Language design and implementation indeed absorbed more project resources than was anticipated. However, the benefits of a well-designed, high-level language are being felt in both coding and validation.

The technique of structured programming has been used successfully in the implementation of several systems.[2,28,29] We have found that the use of structured programming eases the transition from design to coding, and facilitates attempts to demonstrate the correctness of the system.

PROJECT MANAGEMENT

Our goal in Project SUE has been not only to build an operating system, but also to learn about the process of building operating systems. For this reason, we have generated extensive documentation of what the system design is and how it came to be that way. A large (and growing) workbook incorporates project history, project status, problems as they are discovered, solutions as they are proposed, and decisions as they are made. The development of this paper has been based on material contained in the project workbook.

At intervals of about four months, we have written project evaluation reports (Checkpoint Reports). At each checkpoint, we have thought about how well we are progressing, how well we are fulfilling our goals, and whether some redirection of the project is needed. In the narrow view, Checkpoint Reports interrupt our technical progress for periods from three days to three weeks. In broader perspective, Checkpoint Reports have forced us to periodically reevaluate our goals and priorities. Without scheduled Checkpoints, it is unlikely that we would devote enough attention to these topics.

Technical decisions have been made in a democratic way among as many as six people. Democracies tend to progress slowly, but once all parties are convinced of a decision, confidence in the decision is often greater than if the decision had been made by an individual.

Some particularly difficult decisions have been resolved by selecting a reversible decision. The questions could not be completely investigated in the time we were willing to keep a decision pending, so we assured ourselves that the decision taken would not have such broad impact that the decision could not be reversed with reasonable effort.

Although we intended to use "existing technologies", design time, *not* programming time, has been our scarce resource. Two reasons for this have been that most of the designers could not devote full time to the project, and that the system language speeds programming.

CURRENT STATUS

This paper is based on our experience during the first fifteen months of Project SUE. As of July, 1972, we have designed a systems implementation language and implemented a subset sufficient for developing the SUE Kernel and Nucleus. The system structure and all mechanisms for process interaction have been designed and their manner of use documented. The primitive operations provided by the Kernel are designed and are being implemented. Some Kernel modules have been demonstrated correct. Most Nucleus processes are designed and are being implemented. We are encouraging students to create diverse subsystems to be run under the SUE Nucleus.

CONCLUSION

We started Project SUE with the intention of building a reliable, hierarchical, extensible system in which no distinction is made between "user" process and "system" processes. We needed a compatible set of convenient, structured mechanisms for control, communication, authorization, and accounting. We knew of "existing technologies" for handling each of these problems individually, but not of a unified set of mechanisms which treated all the problems. We underestimated the conceptual design effort involved in modifying the existing technologies to make them mutually compatible and appropriate for an extensible system. Most of the systems from which we have drawn ideas were successful at least in part due to their limited goals. We have slowly become aware that our original goals were very ambitious.

A notable change in our approach has occurred since the start of the project. Initially, we designed the most general mechanisms which were implementable. Recently, we have designed the most restricted mechanisms which would satisfy our needs. Partly this is because we now have a much sharper picture of what our needs are. But, also, it reflects a trend toward practicality.

The change in approach can be observed in several areas. The System Language was fully designed early in the project. It has become apparent that much of the generality of the language was costly to implement without contributing greatly to the goals of the project. Early in the project, mailboxes and capabilities in their general sense appealed to us. Both are flexible, powerful, and expensive mechanisms. Recently, we have realized that we can make the system structured and understandable, by using capabilities in a more constrained manner, and using a more restrictive communication mechanism.

It is important to the future of the "Theory of Operating Systems" that new work make use of the knowledge of previous successes and failures in the area. It is also important to test in full-scale systems the adequacy of ideas presented initially at conceptual or philosophical levels. Project SUE has attempted to do both. Frequent introspection has also allowed us to extract some knowledge of designing from our process of design.

ACKNOWLEDGMENTS

Project SUE is supported by the National Research Council of Canada through the Computer Systems Research Group at the University of Toronto.

REFERENCES

1 E W DIJKSTRA
Cooperating sequential processes
in *programming languages* (ed. F GENUYS) Academic Press 1968 pp 43-112

2 E W DIJKSTRA
The structure of the T.H.E. multiprogramming system
Communication of the ACM Vol 11 No 5 1968 pp 341-346

3 B W LAMPSON
Dynamic protection structures
Proceedings of AFIPS FJCC Vol 35 1969 pp 27-38

4 P BRINCH HANSEN
The Nucleus of a multiprogramming system
Communications of the ACM Vol 13 No 4 1970 pp 238-241

5 J B DENNIS E C VANHORN
Programming semantics for multiprogrammed computation
Communications of the ACM Vol 9 No 3 1966 pp 143-155

6 F J CORBATO V A VYSSOTSKY
Introduction and overview of the MULTICS system
Proceedings AFIPS FJCC Vol 27 1965 pp 185-196

7 J H SALTZER
Traffic control in a multiplexed computer system
MAC-TR-30 MIT 1966

8 J W ATWOOD et al
Project SUE status report
CSRG-11 Computer Systems Research Group University of Toronto 1972

9 J W ATWOOD B L CLARK M S GRUSHCOW R C HOLT J J HORNING K C SEVCIK
Proceedings of session '72
Individual papers Canadian Information Processing Society Conference Montreal 1972

10 J J HORNING B RANDELL
Process structuring
CSRG-15 Computer Systems Research Group University of Toronto 1972

11 G H MEALY B I WITT W A CLARK
The functional structure of OS/360
IBM Systems Journal Vol 5 No 1 1966 pp 2-51

12 A N HABERMANN
Prevention of system deadlocks
Communications of the ACM Vol 12 No 7 1969 pp 373-385

13 R C HOLT
On deadlock in computer systems
CSRG-6 Computer Systems Research Group
University of Toronto 1971

14 A SHOSHANI E G COFFMAN
Prevention, detection and recovery from system deadlocks
Technical Report 80 Dept of Electrical Engineering
Princeton University 1969

15 K CORBIN et al
A software laboratory preliminary report
Carnegie Mellon University 1971

16 Y VERNER
On process communication and process synchronization
Dept of Computer Science
University of Toronto 1971

17 C A R HOARE
Towards a theory of parallel programming—a preliminary draft
Queen's University Belfast 1971

18 R S FABRY
Preliminary description of a supervisor for a machine oriented around capabilities
ICR Quarterly Report No 18 Institute for Computer Research University of Chicago 1968

19 G S GRAHAM
Protection structures in operating systems
Dept of Computer Science
University of Toronto 1971

20 W A WULF et al
BLISS reference manual
Computer Science Dept
Carnegie Mellon University 1970

21 N WIRTH
PL360—A programming language for the 360 computers
Journal of the ACM Vol 15 No 1 1968 pp 37-74

22 N WIRTH
The programming language PASCAL
Acta Informatica Vol 1 No 1 1971

23 R D BERGERON J GANNON A VAN DAM
Language for systems development
SIGPLAN Notices Vol 6 No 9 1971

24 W M McKEEMAN J J HORNING D B WORTMAN
A compiler generator
Prentice Hall 1970

25 B L CLARK
The design of a system programming language
Dept of Computer Science
University of Toronto 1971

26 B L CLARK J J HORNING
The system language for project SUE
SIGPLAN Notices Vol 6 No 9 1971

27 COMPUTATION CENTRE
Efficient production of large programs
Proceedings of International Workshop
Polish Academy of Sciences Jablonna Poland 1970 p 81

28 F T BAKER
Chief programmer team management of production programming
IBM Systems Journal Vol 11 No 1 1972

29 B H LISKOV
The design of the VENUS operating system
Communications of the ACM Vol 15 No 3 1972 pp 144-49

System quality through structured programming

by F. T. BAKER

IBM Corporation
Gaithersburg, Maryland

INTRODUCTION

Experience in development and maintenance of large computer-based systems for government and industry has led the IBM Federal Systems Division to the formulation of a new approach to production programming. This approach, which couples a new kind of programming organization (a Chief Programmer Team) with formal tools for using structured programming in system development,[1] was recently applied on a contract with The New York Times for an online information system. Compared to experience on similar contracts in the past, the approach resulted in increased programmer productivity coupled with improved quality. An earlier paper[2] describes the approach in detail and gives productivity measures in a form which should allow comparability to other systems. Following a brief description of the system and a review of the approach, this paper discusses the quality of the system as observed during a thorough acceptance test and in the initial period of operation following its delivery.

THE INFORMATION BANK SYSTEM AND ITS DEVELOPMENT

The New York Times Information Bank is an on-line system which will eventually replace the clipping file (morgue) now used by the Times to provide background information for articles being written. An inquirer may interact with the on-line system to select index terms, specify document parameters (e.g., date of publication, section of the paper), and view article abstracts until he has identified those articles relevant to his immediate needs. Reporters and editors at the Times do this by means of an IBM 4506 Digital TV display unit which can display either text transmitted from the IBM System/360 Model 40 Central Processing Unit or images from standard TV cameras, and transmit text to the System/360 from a standard keyboard. They may also view the full original articles, which are stored in a microfiche retrieval device containing TV cameras capable of being switched to the IBM 4506's under System/360 control.

While editorial support is the main purpose of the system, a number of other features are provided. In addition to the 40 terminals mentioned above, another 24 IBM 4506 units without article viewing capability are interfaced to the on-line system for use by indexers keying index terms, abstracts and document parameters for eventual entry into the system files. The Times is marketing the retrieval service, and up to 500 remote terminals may be added to the system. (While remote users cannot view the articles on their terminals, they can view the abstracts, which provide information sufficient to permit retrieval of the articles from back issue files or from microfilm.) The on-line system (the "Conversational Subsystem") is supported by other subsystems which provide the security data and interactive message texts used by it, edit the keyed indexing data, maintain the system files, print abstracts and clipping references so that users may receive hard copy, log all major interactions with the system and maintain and print statistics on its use. All programs operate on the 360/40 under control of the Disk Operating System.

The system was developed by a Chief Programmer Team, a functional programming organization similar in concept to a surgical team. Members of the team are specialists who assist the Chief Programmer in developing a program system, much as nurses, anesthesiologists and laboratory personnel assist a surgeon in performing an operation. A team is organized around a nucleus of a Chief Programmer, a Backup Programmer and a Programming Librarian. The Chief Programmer is both the prime architect and the key coder of the system. The Backup Programmer works closely with the Chief to design and produce the system's key elements, as well as providing essential insurance that development can continue should the Chief leave the project. The

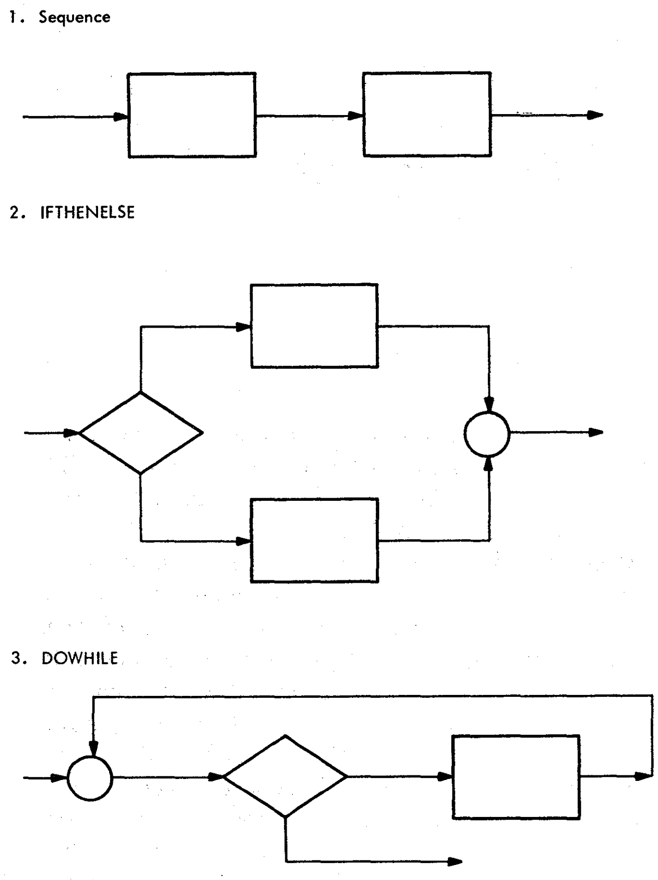


Figure 1—Progressions allowed in structured programming

Programming Librarian is responsible for maintenance and operation of a program library system used to keep all system programs and data both internally in machineable form and externally in well-organized, highly readable form. This Team nucleus, usually assisted by a systems analyst, designs and begins development of the system. The Team is then augmented by additional programmers who produce the remainder of the code under the close supervision of the Chief and Backup Programmers. "Egoless programming,"[3] featuring careful code review by team members other than the original programmer, is practiced throughout.

In addition to the functional organization and the enhanced cooperation fostered by the program library system, the Team operates in a highly disciplined fashion using principles of structured programming described by Dijkstra[4] and formalized by Mills.[5,6] These couple a top-down, evolutionary approach to systems development with the application of formal rules governing control flow within modules. In the top-down approach a nucleus of control code is written and debugged first. Function code is then written incrementally and added to the already operational system. This approach eliminates the need for throwaway drivers and reduces integration problems typically encountered at the end of a project. It also improves reliability because code is debugged within the actual system and because major portions of the system, including critical control code, are operational during almost the entire development period. The rules governing control flow are a consequence of a program structure theorem proved by Böhm and Jacopini.[7] This states that any proper program—a program with one entry and one exit—can be written using only the programming progressions illustrated in Figure 1.

Application of these rules permits a program to be read from beginning to end with no control jumps. It therefore simplifies testing and greatly enhances the visibility and understandability of programs. Finally, it supports the writing of program modules in top-down fashion by enhancing the ability to write and debug control code before adding function code.

DEBUGGING EXPERIENCE

Throughout the development of the system, progress was noticeably enhanced due to the use of structured programming and the library Although no statistics on number of errors or number of runs per module were kept, it was apparent from a qualitative standpoint that both were significantly reduced when compared to similar systems on which team members had previously worked. In a number of cases, program nuclei consisting of two to four hundred source statements ran correctly the first time. In all cases, debugging was clearly faster. Identification of paths to be tested was greatly facilitated by the use of only those formalized control structures permitted by our structured programming conventions.

ACCEPTANCE TEST EXPERIENCE

The system was developed in two major steps. To allow the Times to prepare data for the system files and for debugging and testing of the on-line system, the File Maintenance Subsystem was developed first. Following delivery of that subsystem, the Conversational Subsystem and the rest of the supporting subsystems were developed.

Rigorous and extensive formal testing was performed prior to acceptance by the Times of each of these major phases of the system. For each phase, a test plan was developed jointly by IBM and the Times. Each plan was designed specifically to test all functions included in that phase and was derived principally from the detailed functional specifications agreed upon by the two parties. Data to test these functions were then prepared exclusively by the Times, and these acceptance tests were conducted by the Times with IBM personnel in attendance. All the functional tests were rerun after all problems identified had been corrected, so that corrections could not have undetected effects on parts of the system already tested.

The File Maintenance Subsystem contained 12,029 lines of source code (about 14 percent of the overall system). The test plan for it contained tests for 171 separate functions required in creating and maintaining system files. Acceptance testing lasted a week and also covered all operational aspects of the subsystem, including the elaborate backup and recovery procedures incorporated to ensure preservation of the valuable data. Listings and hexadecimal dumps of all files were made and checked to ensure compliance with predicted file content and specified formats. No errors at all were detected during any of the testing of the File Maintenance Subsystem.

Acceptance testing of the Conversational Subsystem, which contained 38,990 lines of source code (about 47 percent of the overall system), was carried out during a five-week period. The first two weeks were devoted to single-thread (one user signed on from an IBM 4506) testing of the 286 separate functions itemized in the Test Plan. Seventeen errors were detected during this testing, all in the interaction processing modules and none in the time-sharing control program.

Following the single-thread testing, multiple-thread testing was conducted. All the previous tests were repeated with multiple users executing them asynchronously from IBM 4506's. No additional errors were discovered during this testing. Finally, the tests were repeated a third time from IBM 2740 and IBM 2265 terminals, serving to test the remote terminal handling features of the system. While all function was verified to be identical to that observed using the IBM 4506's, three errors were detected in the control program. These all had to do with handling of unusual types of transmission errors on remote lines.

Finally, "Free-form" testing allowed for several periods of retrievals by typical users under conditions when any errors or anomalies detected could be carefully recorded and analyzed. No errors were detected during this type of testing. In addition to the formal acceptance testing, system performance was measured to compare normal and peak load performance to a set of performance goals specified by the Times. Even though the system was operating on an IBM 360/40 with three disk drives, instead of the IBM 360/50 with seven drives which had been proposed and accepted on the basis of the performance goals, the goals were still met.

In all, twenty errors were discovered in the Conversational Subsystem during the five weeks of testing. Only two of those caused abnormal termination of the system; in other words, most of the coding errors were of such a nature that the system continued to function even though output was incorrect. Also, only nine represented bugs in the usual sense; the remaining eleven errors represented functions which had not been incorporated into the coding, or coding which performed as we expected but not as the Times desired. It is also of interest to note that twelve of the errors were in code written during the last two months of the nine-month coding period, and all were in code written during the last four months.

Acceptance testing of the Data Entry Edit Subsystem, which contained 13,421 lines of source code (about 16 percent of the overall system) was carried out during the third week. Pre-defined entries were keyed to test all identified features of this subsystem. However, due to pressure of other duties on the part of the indexers, little free-form testing was conducted. One error (misinterpreted function) was detected in this subsystem as a result of the formal tests.

The five other supporting subsystems, containing 18,884 lines of source code (about 23 percent of the overall system), primarily prepare files and tables for use by the Conversational Subsystem and produce listings, logs and statistical reports on the basis of outputs from it. Because of the variety of conditions required for and ensuing from these tests, it was agreed that the smaller subsystems would be sufficiently tested without the need for additional data. These subsystems were run on a regular basis during the five weeks of acceptance testing, and no errors were detected in any of them.

The complete system contained 83,324 lines of source code. Table I summarizes the total of twenty-one errors found during formal and free-form acceptance testing of the system. In the tables, "incorrect function" refers to code which operated improperly; "omitted function" refers to specifications not implemented; and "misinterpreted function" refers to code which did not perform precisely the functions specified.

OPERATIONAL EXPERIENCE

The File Maintenance Subsystem was delivered in June, 1970. It was used during 1970 and early 1971 to build files for the acceptance testing described above. Beginning in November, 1971, it has been in use on a

TABLE I—Errors Identified During Acceptance Testing

		Error Type			
Subsystem	Source Lines	Incorrect Function	Omitted Function	Misinterpreted Function	Total
File Maintenance	12,029	0	0	0	0
Conversational	38,990	9	8	3	20
Data Entry Edit	13,421	0	0	1	1
Other	18,884	0	0	0	0
Total	83,324	9	8	4	21

daily basis to add to the files new data keyed by the indexers and several years of past data converted from tapes used to publish The New York Times Index. Only two errors have been discovered in this subsystem, neither of which affected the data base. One of these involved incorrect function and the other misinterpreted function.

The Conversational Subsystem was delivered in June, 1971. It was used for experimental and demonstration purposes until November, 1971. Since that time it has been operational eight hours a day for online indexing and for inquiries designed to ensure the consistency of the operational files now being constructed. A total of seven errors have been discovered since delivery. Only one of these resulted in abnormal termination of the system, and this was due to lack of any capability in the System/360 Disk Operating System to handle the particular file error condition which caused it. (Additional application coding was added to circumvent the possibility of this error occurring again.)

The Data Entry Edit Subsystem was also delivered in June, 1971, and became operational on a daily basis in November, 1971. It had the least formal testing of any of the subsystems and has had a number of extensions made to it since delivery. Sixteen errors have been identified in this subsystem.

TABLE II—Errors Identified During Operation

		Error Type			
Subsystem	Source Lines	Incorrect Function	Omitted Function	Misinterpreted Function	Total
File Maintenance	12,029	1	0	1	2
Conversational	38,990	4	3	0	7
Data Entry Edit	13,421	8	5	3	16
Other	18,884	0	0	0	0
Total	83,324	13	8	4	25

The five other supporting subsystems have been used on an intermittent basis since their delivery in June, 1971. No errors have been detected in any of these during that period.

Table II summarizes the operating experience to date which has resulted in a total of 25 errors being identified, only thirteen of which involved incorrect function. This represents about three errors per 10,000 lines of code, a result which informal comparisons suggest is substantially better than average. From another standpoint, there was about one error for each five man-months of effort on the project. In fact, the programs written by the Chief and Backup Programmers had about one error per year of effort on their parts.

Consequently, initial operation has been very smooth. The important Conversational Subsystem has only suffered one abnormal termination due to an error in thirteen months of experimentation and operation; the other five errors prevented a single user from completing an inquiry or entering indexing data but permitted continued operation. To the best of our knowledge, no errors have been created in the files during two years of operation of the File Maintenance Subsystem. The experience with the Data Entry Edit Subsystem has not been as good, and it has suggested some changes in procedure discussed below.

CONCLUSIONS

Structured programming, and the organization and tools used to achieve it, were key factors in developing this kind of system. The fact that most of the errors encountered during acceptance testing were in code written during the last two months tended to confirm our expectations that the longer period of operation permitted by the top-down approach would lead to a more reliable system. The application of the program structure rules made it thoroughly practical for programmers to read, check and criticize each other's code and nearly eliminated the need for flowcharts as a means of communication. The Chief Programmer and Backup Programmer together reviewed much of the code on the project, particularly that of the more junior members of the team. This ensured that specifications and standards were being adhered to and that code would function as intended. Numerous problems were identified by code reviews which would otherwise have led to problems later.

The program library system used was also a major factor in improving quality. Ensuring that up-to-date versions of programs and data were always available reduced problems frequently encountered due to use of obsolete versions. For instance, when programmers were ready to use an interface, they could directly include the appropriate declarations into their code instead of writing their own version. When the interface changed, it was only necessary to recompile to incorporate a new version into all affected programs. In addition to reducing interface problems, the library system facilitated study of code to allow one programmer to adapt an approach used by another instead of re-creating it. Most importantly, it permitted the ready review and criticism of code by others as described above. As a side benefit, the availability of all this information in usable form reduced the need to get it verbally and thus further reduced errors due to distraction or interruption.

While it was not essential to structured programming, the use of the functional Chief Programmer Team organization had three major benefits in the area of program quality. First, the use of senior people directly in the design and programming process led to a cleaner, more rapidly implemented design. Second, use of a programming librarian to do many of the clerical tasks associated with creating, updating and maintaining programs reduced interruptions and diversions which tend to cause programming errors. Finally, the higher degree of specialization and smaller number of programmers led to a reduction in the number of misunderstandings and inconsistencies.

This project has suggested two areas in which further work needs to be done. First, it may not always be possible to follow a strictly top-down approach in development of a large programming system. If a system organization, viewed as a tree structure, is narrow and tall, then a pure top-down approach may take too much elapsed time to be practical. Second, a more rigorous approach to code review needs to be developed. In retrospect, a number of the problems encountered in the Data Entry Edit Subsystem after delivery were of such a nature that they would probably have been caught earlier if all the code had been read. The Chief and Backup Programmers did much functional coding themselves on the project, but it would probably have been more effective for them to have reviewed more code and written less. This would have reduced productivity slightly but would have eliminated a number of the remaining problems.

While the initial objective of the approach was improvement in production programming productivity, it became apparent that the same methods also resulted in increased quality. Experience gained on this project is leading IBM to more experimentation with structured programming and Chief Programmer Teams, and limited results to date confirm the conclusions reached here.

REFERENCES

1 H D MILLS
Chief programmer teams: Principles and procedures
Report No. FSC 71-5108—May be obtained from
International Business Machines Corporation Federal
Systems Division Gaithersburg Maryland 20760

2 F T BAKER
Chief programmer team management of production programming
IBM Systems Journal 11 No 1 pp 56–73 1972

3 G M WEINBERG
The psychology of computer programming
Van Nostrand Reinhold New York 1971 p 72

4 E W DIJKSTRA
Notes on structured programming
Report No EWD249 Technische Hogeschool Eindhoven
Eindhoven Netherlands August 1969

5 H D MILLS
Mathematical foundations for structured programming
Report No FSC 72-6012—May be obtained from
International Business Machines Corporation Federal
Systems Division Gaithersburg Maryland 20760

6 H D MILLS
Top down programming in large systems
Debugging Techniques in Large Systems
Prentice Hall Englewood Cliffs New Jersey 1971 pp 41-55

7 C BOHM G JACOPINI
Flow diagrams, turing machines and languages with only two formation rules
Communications of the ACM 9 No 3 pp 366-371 May 1966

An application of cellular logic for high speed decoding of minimum-redundancy codes

by K. OHMORI, S. NAITO, T. NANYA and K. NEZU

Nippon Electric Company, Limited
Kawasaki, Japan

INTRODUCTION

In the efforts to improve the total efficiency of computer systems and their applications, more importance is being placed on the qualitative improvement of processing information in these days. The use of "Kanji (Chinese characters)" in the system is becoming one of the topical themes of research and development in Japan. It is expected to considerably improve the communication from machine to man. In the case of Kanji, a character generator with a font capacity greater than 1000 is required. The same requirement might exist also in Western countries, if special fonts for Greek or Roman alphabets, italics, bold face, or special mathematical symbols are necessary. Another example is a computer generated high-speed phototypesetting system.

In the realization of the character generator of a large character set, the problem of storing and retrieving the information of character patterns has to be solved. If all the information of the character patterns were directly stored in the high speed memory, the system would be too high in cost. For example, when each individual pattern is composed of a 24×24 dot matrix, which may be considered to be the necessary and sufficient size for representing a character pattern of good quality, the memory capacity needed for 1000 Chinese characters is 576,000 bits. Therefore, compression of the data to be stored has a practical meaning for reducing the cost of pattern generating systems, if a satisfactory speed and design simplicity of the accompanying decoder can be obtained.

Information theory gives us the basic concept and techniques of the data compression.[1] Minimum-redundancy codes, or Huffman Codes,[2] are the most efficient ones when the statistical property of original data is known.

In the character generation, the encoding speed does not directly affect the performance of the system. On the other hand, decoding speed is essential, because it determines the printing or displaying speed of the output device. In decoding, the trouble is that Huffman Codes, whose length is variable, have to be decoded bit by bit during read out of the data from memory. This has encouraged the development of high speed decoding hardware.

This paper presents a new high speed decoding system consisting of cellular logics which have such merits as high decoding speed, design simplicity and ease of machine fault detection.

ENCODING CHARACTER PATTERNS

As shown in Figure 1, the character pattern for a certain character is defined as a dot matrix, each element of which is given either one of the binary symbols, or in other words, binary states B (black) or W (white). We can easily see that the occurrence of a certain state of any dot in the matrix is not independent of the states of the neighboring dots, because of some specific characteristics of the character patterns.[3] For instance, the probability of a dot being black is very large, if all the neighboring dots are black. This fact shows that the dot matrix has a great deal of redundant information in it.

In order to encode the dot matrix mentioned above efficiently, let's consider a group of dots as shown in Figure 2. We call the group of dots a subpattern, hereafter. A subpattern consisting of a 2×2 dot matrix has 2^4 states.

Which state a subpattern lies in depends considerably on the state of neighboring subpatterns, especially on the ones below and above, or to the right and left. This fact comes from the characteristics that Chinese characters involve many vertical and horizontal straight

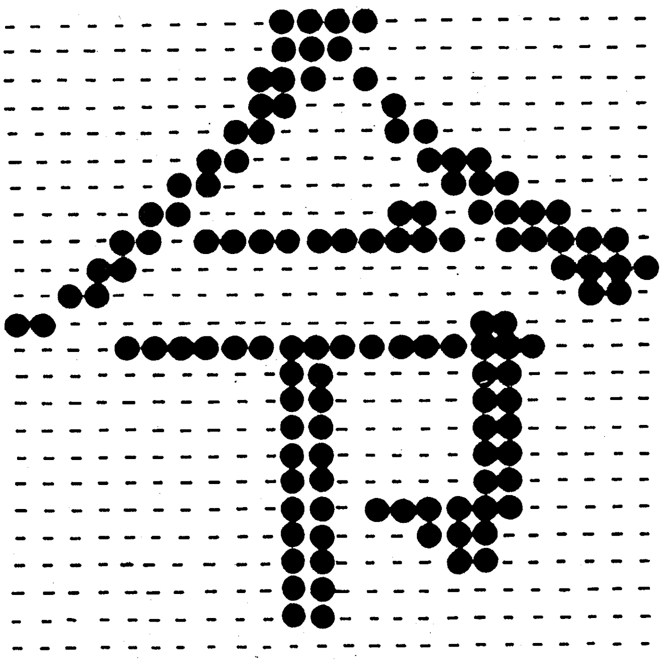


Figure 1—Chinese character pattern consisting of a 24×24 dot matrix

strokes in their patterns. Therefore, considering the conditional probability of the occurrence of a certain subpattern under the condition that the state of subpattern to its left is given, the entropy of the character set, which is the theoretical lower limit of the average code length, possibly decreases as compared with the nonconditional entropy.

For the Chinese character set (the size of the matrix is 24×24 as shown in Figure 1), the average code length for a character is reduced to 300 bits, while in the nonconditional case, the average length is 380 bits. Because 576(24×24) bits for a character is needed without encoding, this means that the memory capacity needed has been reduced to half.

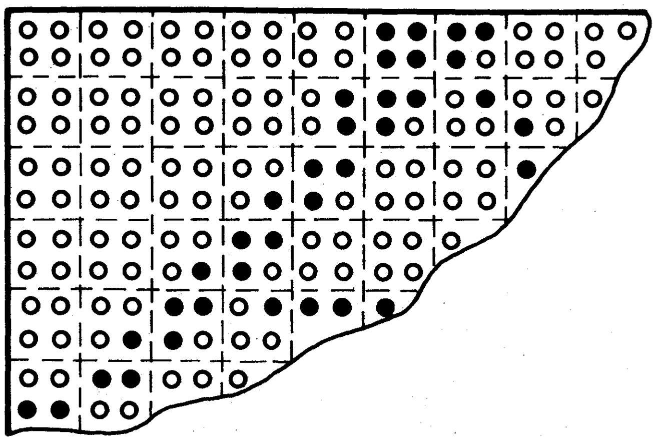


Figure 2—Subpatterns consisting of a 2×2 dot matrix

As each subpattern has 16 states, as mentioned above, 16×16 messages are necessary for encoding the patterns. In other words, 16 Huffman trees each having 16 leaves have to be defined for encoding. Such a quantity of messages or message codes makes it complicated to design high speed decoder circuits. We solved this problem by introducing cellular logics.[4,5]

TOTAL SYSTEM

A block diagram of the total system is given in Figure 3. It primarily consists of code-address converter, memory, parallel-serial converter, decoder, ring buffer register and several asynchronous controllers. The inputs which should be applied to the system are character codes, each of which is generated either by CPU, key board, card-reader or tape-reader, and designates a corresponding character pattern. The outputs of this system are the coordinate signals X, Y and a pattern signal Z, which should be applied either to CRT, printer or any other display devices.

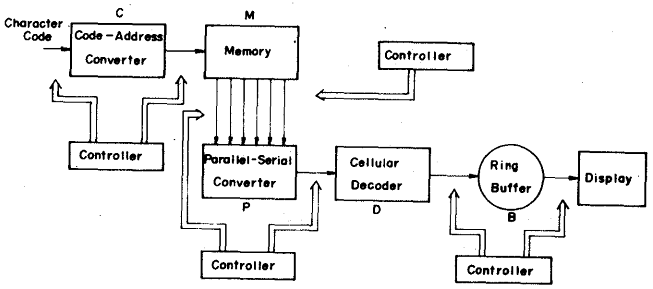


Figure 3—Block diagram of pattern generating system

Now, we will outline the behavior of each component in the system and the signal flow between them by means of Figure 3.

The encoded character pattern data are stored in the memory M. Since the stored data of the character patterns are composed of variable length code words, the addressing method required is somewhat an elaborate one.

Generally, character codes don't have any information related to the code word length of the encoded data or to the original character pattern. Therefore, when the pattern generator receives a code which designates a specific character pattern, the code must be transformed by some way into an address which indicates the location where the data corresponding to the pattern is stored.

Concerning the data-storing and its addressing, there are two ways, in general to approach it. One represents a viewpoint wherein the data should be stored in the main memory of a computer, and the addressing is executed by some software. The other viewpoint is where a pattern generator should have its appropriate memory all by itself and where code-address conversion is executed by hard wired logic. In this paper, the latter has been adopted because of its processing speed and the saving of CPU time. The code-address converter C in Figure 3 performs such a function as mentioned above, and its details will be described in the following section.

As one can easily see from the preceding section, the decoder D processes the encoded data bit by bit, while the encoded data must be read out from the memory M word by word. So, a register P for the parallel-serial conversion is required. This is composed of identical components which are called two-dimensional cells. The modes of the information transmission along the X-axis and the Y-axis are quite symmetric. This function makes it possible to receive parallel signals and put out serial signals without losing any bit-time as will be described in detail later.

Now, recalling that Huffman codes are assigned for ordered pairs of subpatterns and that each subpattern has 16 kinds of states, it is clear that the decoder D must be composed of 16 tree networks, each one of which is a practical realization of a decoding system graphically represented by a binary tree proper to a certain given state of the preceding subpattern of an ordered pair. Each tree has a different structure from any other tree and has 15 nodes in order to branch into 16 subpatterns. Considering that, at each node of the 16 trees, an identical component, or cell, is placed for branching elements, the decoder fundamentally consists of 240 cells. The actually realized system, however, consists of only 180 cells by reduction, utilizing the concept of node equivalence.

Considering again that the encoded data for the pattern information consists of the variable length codes, we can see that the decoder D puts out 2×2 dot patterns at random intervals. Therefore, in order to display a character pattern by arranging 144 subpatterns, the output of the decoder must be transformed into a serial bit signal at regular intervals according to the scanning mode of the display device. The ring buffer register B has such functions of signal transformation. It consists of circularly arrayed 96 bits memory cells and two endless shift registers. A detailed explanation will be also given in the next section.

Finally, the signal flow of the system is summarized as follows.

(1) A character code is received by C and converted into the address corresponding to the head of the data-storage location in M.
(2) Encoded data are read out word by word and applied to D bit by bit in serial through P.
(3) The data is decoded into subpatterns consisting of 4 bits-binary signals at random intervals.
(4) The subpatterns are arranged into a full dot pattern of 24×24 by B and signal Z is obtained at regular intervals, being accompanied with the coordinate signals X and Y.

Concluding this section, the interconnection between each component explained above are supervised in parallel by several controllers consisting of asynchronous sequential circuits.

COMPONENTS

Code-address converter and memory

The pattern data are stored according to the order of the character codes. As we have seen, the encoded data do not require any punctuation mark between adjacent data because of a uniqueness of the decoding of Huffman codes. Therefore, in order to save memory capacity, the head of each record (pattern data for a character) is superimposed on the tail of its preceding one, if possible, as shown in Figure 4. Consequently, the length of each record is variable, and its distribution is approximately considered to be Gaussian with mean length of 300 bits. The starting location of each record is obtained by means of linear regression as shown in Figure 5.

For efficiency of the regression, all records for a full

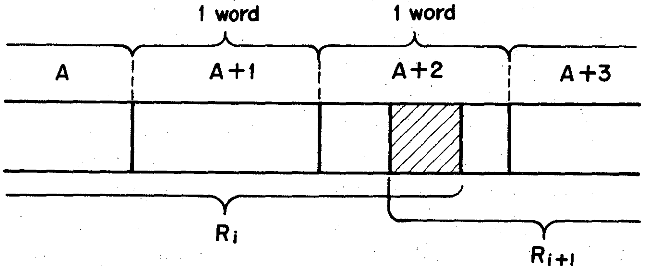


Figure 4—Data structure in memory

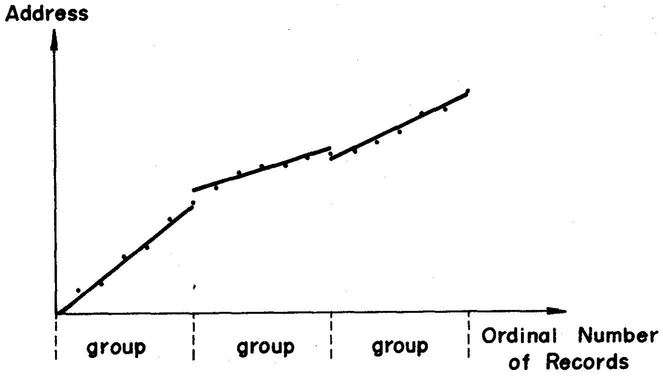


Figure 5—Linear regression for addressing

character set are partitioned into several groups. A regression line $y=ax+b$ is chosen for each group so as to minimize the maximal deviation Δ from the line in the group. Then, only if regression coefficients a and b for each group and a deviation Δ for each record are stored, the address A of the starting location of any record can be obtained by the relation:

$$A=aN+b+\Delta$$

where N denotes the ordinal number of the record.

After all, the code-address converter consists of a multiplier, controller and some registers.

Parallel serial converter

The parallel serial converter used in the system is a kind of shift register composed of 20 cells, each of which has a functional line, 2 input lines and a output line. The functional line selects one of two input lines (X input and Y input) and the cell stores the value of the selected input line when the shift pulse comes in. At the same time, the stored value at the last shift pulse is sent into the output line.

The block diagram of the parallel serial converter is shown in Figure 6. In the diagram, X_i and Y_i are the input lines of the cell. Z_i and f_i are output line and functional line, respectively.

Usually, the value of functional line is set to select the X input. Therefore, the converter operates as an ordinary shift register. However, the value of functional line changes and selects the Y input line when the shift pulse counter indicates 19. Then, at the time of next shift pulse, the cells receive 20 bit signals in parallel from memory and the last bit is sent out simultaneously into Z_1.

Decoder

A cell for the tree network capable of decoding serial codes should have the following 3 functions.

1. To select one of two output lines according to selection signals.
2. To register the input signal, until the next selection signal comes.
3. To send an output signal to a selected output line.

The realized cell is shown in Figure 7. In the circuit, the Master Slave J-K flip flop is utilized as the memory element.

The number of nodes in a tree structure is $N-1$ when the number of messages which are distinguishable from each other is N. In our decoder, 16 code systems should be decoded and each code system has 16 messages; therefore, 240 cells are needed fundamentally. However, there exist some nodes from which the tree branches similarly and which lead to the same output signals. These nodes are considered to be equivalent in the sense of Moore machine and are able to be reduced to one node. As the result of the reduction, the realized decoder has 180 cells.

Figure 8 shows the state transition of the decoder consisting of 16 binary trees.

The 16 kinds of subpatterns are numbered according to the following rule:

(1) Using the 2×2 dot matrix representation, regarding white dots and black dots as "0" and "1" respectively, rewrite the matrix in binary form.
(2) Rewrite the 2×2 matrix in a single 4-bit row

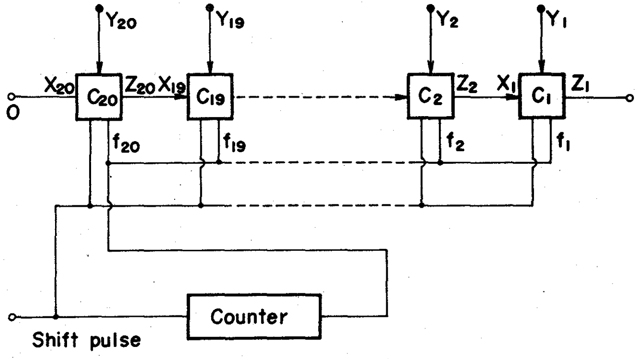
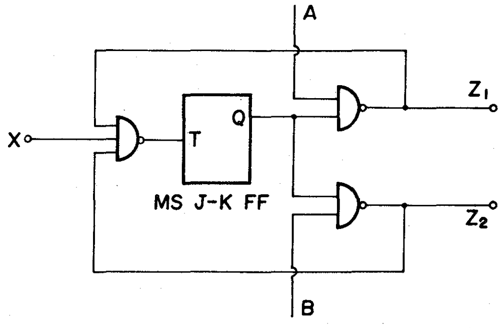


Figure 6—Parallel-serial converter



X : input line
A,B : selection lines
Z_1, Z_2 : output line

Figure 7—Cell in decoder

form by placing the second row in the first two bit positions.

(3) Next, add "0001" and assign the decimal number equivalent to the sum.

Thus, the subpatterns are represented by the numbers 1 to 16.

The initial state is set in cell A1 and the output of state A1 is set at subpattern No. 1. When input code "1" is received by cell A1, cell A2 fires. On the contrary, if the input code "0" is received, cell A1 merely continues to fire. When cell A2 is fired and it receives input code "0," cell A3 fires, and finally, if cell F1 or P1 is fired according to the input codes "0" or "1," subpatterns No. 6 or No. 16, respectively, will be generated.

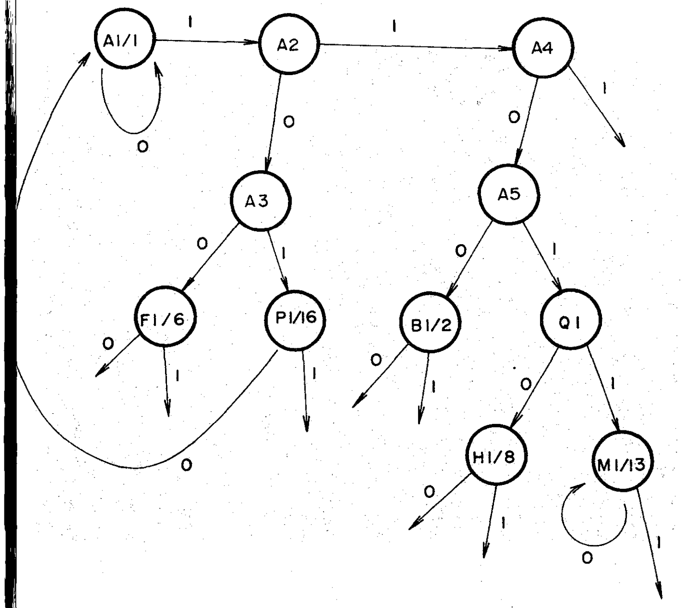


Figure 8—State transition diagram of the cellular decoder

Since our original Chinese character was partitioned into a 12×12 array of subpatterns and our processing scheme is concerned with ordered pairs of subpatterns, this routine is iterated until 144 subpatterns have been decoded, that is to say until completion of the generation of the Chinese character.

The decoder consisting of cellular logics of tree structure is well suited for easy fault detection. Because the tree network essentially has no feedback loop, such difficulties as frequently occur in sequential logic circuits can be eliminated. Practically speaking, by applying each code sequence systematically to the input terminal, observing whether the corresponding output terminal fires or not and by combining the firing terminal cells with the nonfiring ones, any faulty cells in the tree network are sure to be detected. The procedure of fault detection can be automatically executed.

Ring buffer register

Figure 9 shows the principle of the ring buffer register. It consists of circularly arrayed 96 bits memory cells and two endless shift registers. One of the two shift registers is the address register *W*, for the write mode and the other, *R*, for the read mode.

For convenience in explanation, let's number each memory cell from 1 to 96 as shown in Figure 9, and similarly, number each dot of 24×24 dot matrix in such way as shown in Figure 10.

The address register *W* simultaneously designates four locations of the circularly arrayed cells for writing of the information of 2×2 dot matrix, and is initially set at cell 1, cell 2, cell 25 and cell 26. The address register *R* is for reading the pattern information registered in the suitable locations, and is initially set at cell 0 (= cell 96).

Now, the first output of the decoder must be a subpattern (d_1, d_2, d_{25}, d_{26}), while the scanning mode of the dot matrix on the display device is given by,

$$d_1, d_2, d_3, \text{-----}, d_{24}, d_{25}, d_{26}, d_{27}, \text{-----}, d_{576}.$$

Therefore, the bit signals d_{25} and d_{26} must be stored for a time in ring buffer register, so that the locations designated by *W* are required to be located in cell 1 for d_1, cell 2 for d_2, cell 25 for d_{25} and cell 26 for d_{26}. Then after the data of 4 bits has been registered in the corresponding cells respectively, the address register *W* shifts by two addresses and waits for the next output

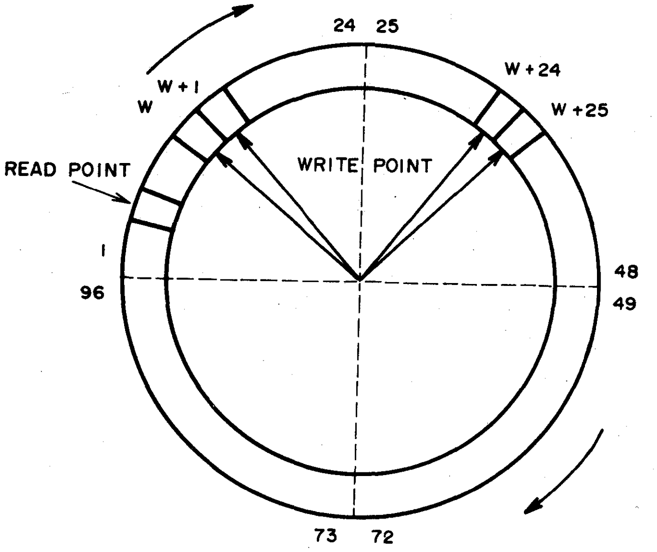


Figure 9—Diagram illustrating the behavior of the ring buffer register

(d_3, d_4, d_{27}, d_{28}). Thus, four locations designated by W shift by two addresses whenever the 2×2 dot matrix is put out from the decoder.

When the data of 12 subpatterns for two rows of the 24×24 dot matrix has been registered, another address register R starts to shift bit by bit for reading the pattern information just registered, synchronizing with the horizontal scanning on the display device.

The circulation of both address register W and R along the memory cells are quite independent of each other, because the 2×2 dot pattern is put out at random intervals in spite of the regular intervals of display scanning.

Summarizing, the ring buffer register has the following two functions.

(1) To transform a signal generated at random intervals into one generated at arbitrarily required intervals.
(2) To transform parallel bit signals into serial bit signals.

Controllers

All the controllers in Figure 3 consist of asynchronous sequential circuits and their jobs of supervising each interconnection between components are executed completely in parallel by each corresponding controller.

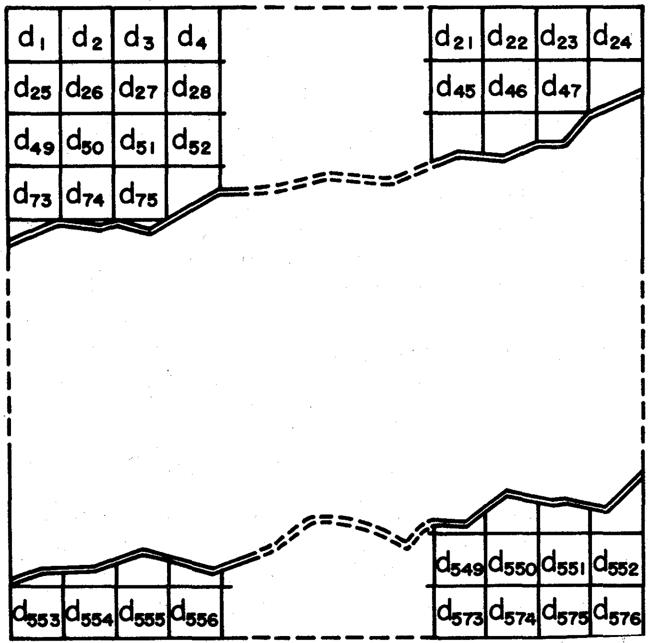

d_1 d_2 d_3 d_4
d_{21} d_{22} d_{23} d_{24}
d_{25} d_{26} d_{27} d_{28}
d_{45} d_{46} d_{47}
d_{49} d_{50} d_{51} d_{52}
d_{73} d_{74} d_{75}
d_{549} d_{550} d_{551} d_{552}
d_{553} d_{554} d_{555} d_{556}
d_{573} d_{574} d_{575} d_{576}

Figure 10—Dot matrix on display

RESULTS

The pattern generating speed of this system is 8000 characters per second, and the memory capacity on an average requires 300 bits for a Chinese character to be stored by data compression, while the original bit pattern of the dot matrix for a Chinese character has 576

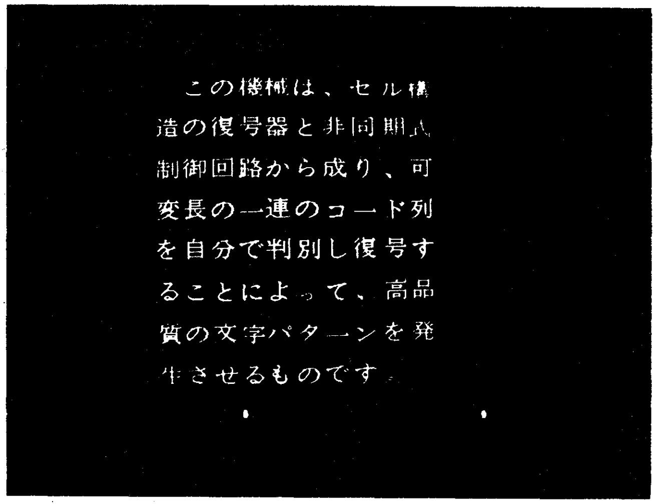

この機械は、セル構
造の復号器と非同期式
制御回路から成り、可
変長の一連のコード列
を自分で判別し復号す
ることによって、高品
質の文字パターンを発
生させるものです。

Figure 11—An example of generated Chinese characters

bits of information. This result means that the memory capacity needed for a pattern's data was reduced to half.

An example of Chinese characters generated by the present system is shown in Figure 11.

CONCLUSION

A high speed decoder consisting of cellular logics has been explained. The decoder has such merits as high decoding speed, design simplicity and ease in detecting faults in the circuits. These merits were confirmed by a practical decoder developed for high speed pattern generation of a large character set. Among the merits mentioned above, the design simplicity may be most essential. The simple and clear representation of a given Huffman tree and simple connection of trees eliminate awkwardness of implementation of Huffman Codes.

Further investigation into the reduction of the number of cells required in the decoder without loss of these merits is expected to be conducted.

ACKNOWLEDGMENT

The authors wish to thank Mr. Yasukuni Kotaka for his kind interest in this work.

REFERENCES

1 C E SHANNON
A mathematical theory of communication
The Bell System Technical Journal Vol 27 No 3 1948

2 D A HUFFMAN
A method for the construction of minimum-redundancy codes
Proceedings of the IRE Vol 40 Nov 9 1952

3 K NEZU
A method for encoding character patterns utilizing mutual information between dots
The Transactions of The Institute of Electronic and Communication Engineers of Japan Vol 55-D No 4 1972

4 M M NEWBORN
A synthesis technique for binary input-binary output synchronous sequential Moore machines
IEEE Transactions on Computer Vol C-17 No 7 1968

5 T F ARNOLD et al.
Iteratively realized sequential circuits
IEEE Transactions on Computer Vol C-19 No 1 1970

On an extended threshold logic as a unit cell of array logics

by RYOICHI MORI

Electrotechnical Laboratory
Tokyo, Japan

INTRODUCTION

Some of our viewpoints on array logics follow.

(a) The unit cell of the array should have as high logical abilities as possible.
(b) The number of interconnecting wires are limited so they should carry as much information per wire as possible.
(c) There is no reason for that the noise immunity inside the array logic must be as high as outside the array, for example, as that required for inter peripherals.

These points suggest the use of multi-valued logic. However, general multi-valued logic lacks some kind of generality or simplicity, for example, 4-valued logic is quite different from 3-valued one, and so on, and practical hardwares have rarely been reported. On the above described standpoint, we have made research on a concept of an extended threshold logic, briefly AETL. Logical theory and practical hardwares have been developed. In short, this AETL presents a much higher logical ability with basically the same hardware compared with usual threshold logic. Short explanations of basic idea, theory, and hardware development follow.

In usual threshold logic, there are input variables x_i which takes a value 0 or 1, and $\sum w_i x_i$ is given multiplying weight w_i and making a linear summation. Then it is compared with threshold t and the output y takes either 0 or 1 depending on the result of the comparison. The threshold t is variable as of W. S. Meisel[1] and is plural as of D. R. Haring,[2] in some papers. There is also a hardware of integrated threshold logic.[3] We won't go further in details as excellent reviews[4,5] have been given.

Unlike this, the basic concept of AETL is as follows. Consider a pair of inputs x_i, x_i'. See which is larger and generate 0 or 1 depending on the result. Multiply it by weight w_i and make a linear summation. And let the result be output y. From another point of view, usual threshold logic performs the linear summation at input side. On the contrary AETL performs it at output side. They may have a question as follows. Outputs of a certain logical circuit are usually given to some inputs of some circuit. If we consider a system comprising a number of logical circuits, isn't it the matter of definition which is the input side (which is the output side)? Isn't the actual hardware the same? The answer has two aspects.

The first thing is that the degree of technique required in manufacturing a hardware is basically the same. The reason is that, when the result of linear summation takes certain value, it is necessary to distinguish this value from the value next to it without fail, regardless the temperature, source voltage, manufacturing variations, aging, etc., so the required accuracy is the same basically.

The second thing is that the two configurations are entirely different in their logical abilities, as explained below. In usual threshold logic, both input x_i and output y are two-valued. The result of the linear summation is multi-valued and contains plenty of information, but it exists inside of unit circuit and is not accessible from outside. Only one binary information, output y, is accessible from outside and other informations are all lost. On the contrary, in AETL, the result of linear summation is the output y which is multi-valued and accessible as the input of arbitrary circuit and so its logical ability is far better.

An arbitrary boolean function can be realized by two AETLs.* Two 3-input-pair AETLs realized a full adder. One 2-input-pair AETL realized a Data F/F. Using 4-input-pair AETL as master and as slave, a 2-dimensional shift register has been realized. Of course all previously known threshold logic can be realized by AETL.** Although AETL is a multi-valued logic, it is

* This means two AETLs realize a hardware of multi-threshold threshold logic.

** In AETL, when x_i' is selected as a constant, output y is the same as $\Sigma w_i x_i$ of usual threshold logic. Let y be the input x_i of the next stage and give t as x_i'. Then the result is the same as the usual threshold logic.

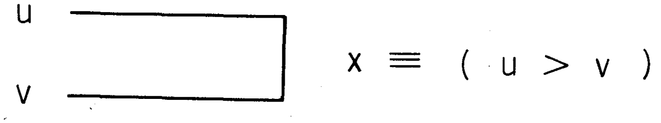


Figure 1—Symbolic expression

different from usual 3-valued or multi-valued logic. The main difference is that AETL is based on the most simple method, that is, the comparison of algebraic magnitudes in operation of multi-valued variables, instead of seeking a basic set of logical operations which realize all logical functions in multi-valued logic. Hardware realizations of basic operation become straightforward by this AETL method. Therefore AETL has made it possible to design 4-valued, 9-valued, and 17-valued logical hardware on the same basis.

SYNTHESIS PROBLEM BY AETL[6]

Definition of AETL

Let's define a notation as follows (Figure 1).

$$(u>v)=\begin{cases}1 & \text{if } u>v\\ 0 & \text{if } u<v.\end{cases}$$

Then, usual threshold logic is defined as follows.

$$y=\left(\sum_{j=1}^{n} w_jx_j>t\right) \qquad x_j=0,1 \quad y=0,1.$$

AETL is fundamentally based on threshold logic. Figure 2 shows AETL's graphic symbol. AETL is defined as follows.

$$y=\sum_{j=1}^{n} w_j(u_j>v_j)=\sum_{j=1}^{n} w_jx_j$$

$$x_j=0,1 \quad y=0,1,\ldots,\sum_{j=1}^{n} w_j$$

where y: output u_j, v_j: input w_j: weight.

AETL has one output. However, logically, AETL generates many outputs comparing y with values t_j.

Let u_j be variable and v_j, t_j be constant. Then Figure 2 is reduced to Figure 3. This restricted usage of AETL is called Multiplex Median Logic, for short, MML.[7,8]

An MML circuit can generate a complete set of median functions for a given weight vector (Figure 3). Here median functions $M_t{}^W$ are defined as follows. For a weight vector $(w_1, w_2, w_3, \ldots, w_n)$,

$$M_t{}^W(w_1x_1, w_2x_2, \ldots, w_nx_n)=\begin{cases}1 & \text{if } \sum_{j=1}^{n} w_jx_j\geqq t\\ 0 & \text{if } \sum_{j=1}^{n} w_jx_j<t\end{cases}$$

where w_j: weight of x_j x_j: 0 or 1 $W=\sum_{j=1}^{n}|w_j|$.

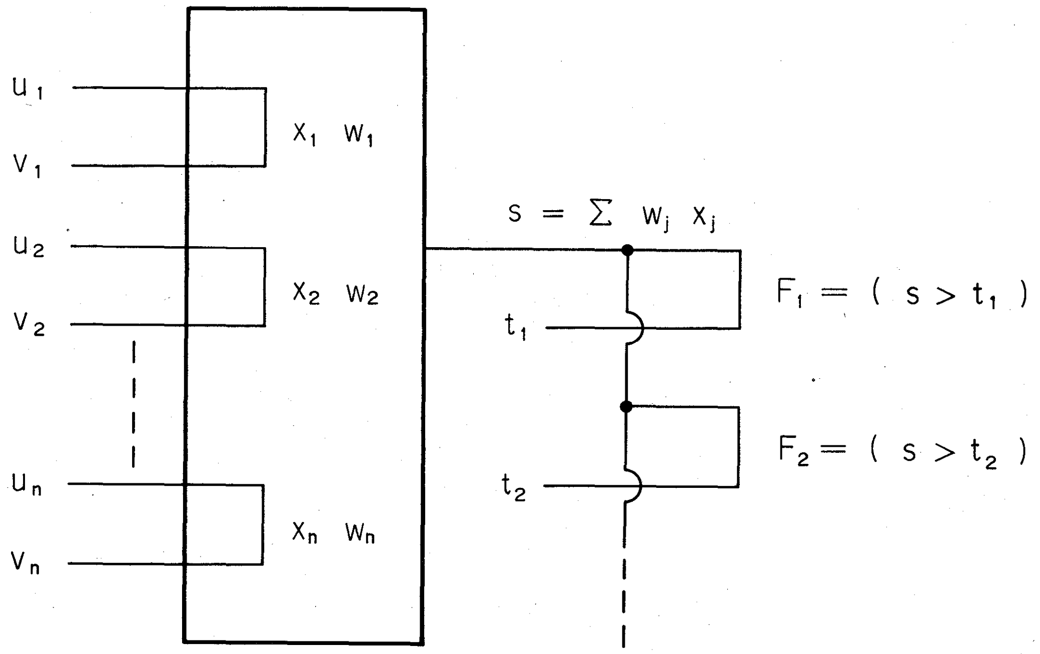


Figure 2—Graphic symbol of AETL

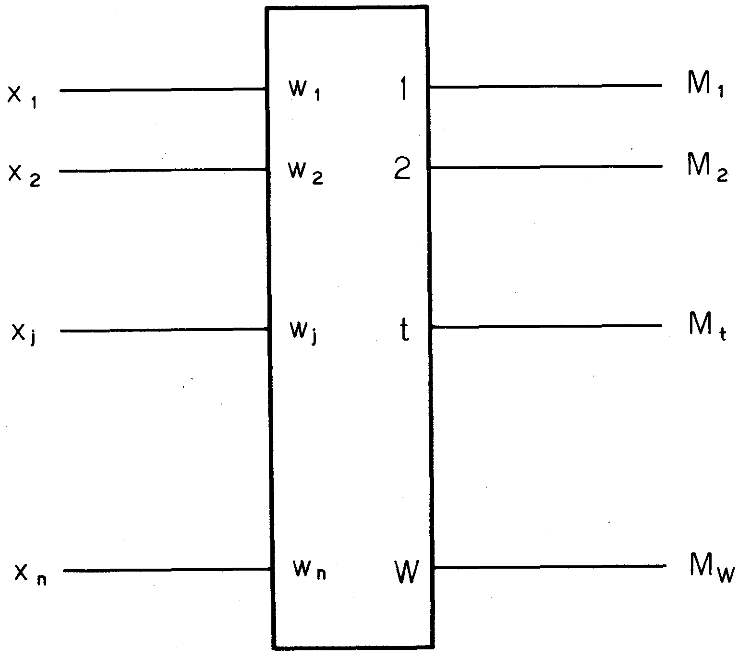


Figure 3—Graphic symbol of MML

Synthesis of logical functions[9,10]

First of all, we give a fundamental theorem concerning synthesis problems by MML.

Theorem 1.

The necessary and sufficient condition that a given n-variable logical function can be realized by two cascaded MMLs as shown in Figure 4 is as follows: If

$$f(\vec{X}) \neq f(\vec{Y}), \quad \text{then } \overrightarrow{WX} \neq \overrightarrow{WY} \text{ for all } \overrightarrow{XY}, \in V^n \quad (1)$$

where w_j is the weight of x_j and $\vec{W} = (w_1, w_2, \ldots, w_n)$. V^n is the whole of n dimensional binary vectors. The weight sum of the second stage MML does not exceed that of the first stage.

When n variables of a function f are partially symmetric and classified into h blocks as $n_1 + n_2 + \cdots + n_h = n$, we say that f has a symmetric pattern $(n_1, n_2, \ldots, n_h)$.

Theorem 2.

When a given function f has a symmetric pattern $(n_1, n_2, \ldots, n_h)$, f can be realized by assigning the weights of the first stage MML in Figure 4 as follows. Let all variables belonging to the kth block have the

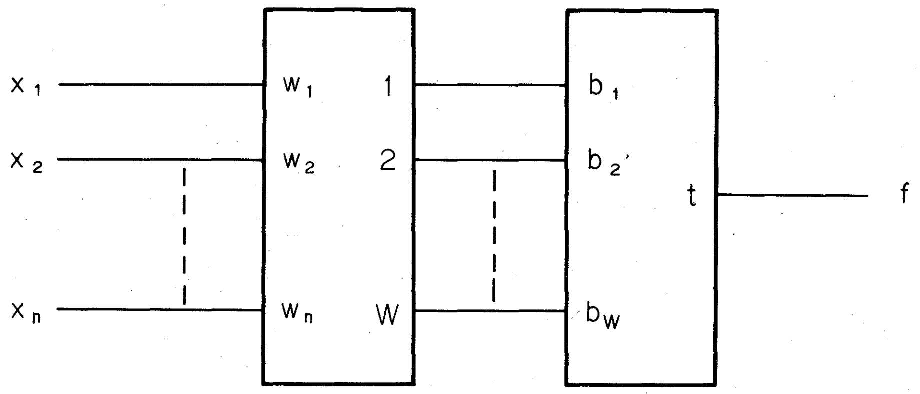


Figure 4—Simple cascade connection

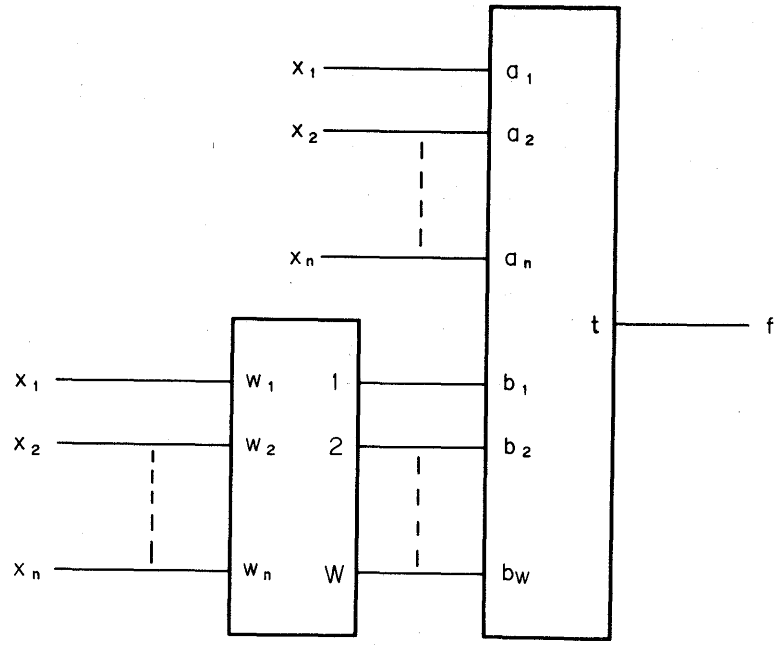


Figure 5—Bypassed cascade connection

same weight u_k, and

$$u_1=1, \quad u_k=\sum_{j=1}^{k-1} u_j n_j+1 \qquad (2\leqq k\leqq h).$$

Then total weight sum W is equal to

$$(n_1+1)(n_2+1)\ldots(n_h+1)-1.$$

W does not depend on the order of assigning the weight for each block.

Lemma 2-1

An arbitrary n-variable symmetric function can be realized by at most two MMLs of which weight sum W is at most n.

For the practical purpose, there is no reason to confine to the case of Figure 4. Figure 5 is the most generalized connection of two MMLs.

Concerning the minimization of W in this case, we show next theorem.

Theorem 3.

In Figure 5, given function f, given weight $\vec{\mathbf{W}}=(w_1, w_2, \ldots, w_n)$ of the first stage, the necessary and sufficient condition for f to be realized is as follows: If and only if, without depending on the parameter h, there exists a vector $\vec{\mathbf{A}}=(a_1, a_2, \ldots, a_n)$ such as $\overrightarrow{\mathbf{AX}}>\overrightarrow{\mathbf{AY}}$ for all $\vec{\mathbf{X}},\vec{\mathbf{Y}}\in C_h, f(\vec{\mathbf{X}})>f(\vec{\mathbf{Y}})$, f can be realized for the given weight $\vec{\mathbf{W}}$, where

$$C_h\equiv\{\vec{\mathbf{X}}\mid\overrightarrow{\mathbf{WX}}=h\}.$$

Minimization of weight sum W

We presented a kind of table look up method. Once we have prepared the table of SWV which means Standard Weight Vector, we can select an optimum weight vector from the table of SWV for a given function, where "optimum" means the minimum weight sum.

The numbers of SWV for 1- to 4-variable functions are given in Table I. As seen from the table, the ratio

n	1	2	3	4
N_1	1	2	5	19
N_2	2	3	14	222

N_1 : Number of standard weight vectors

N_2 : Number of representative functions

TABLE I—N_1,N_2 of n-Variable Boolean Function

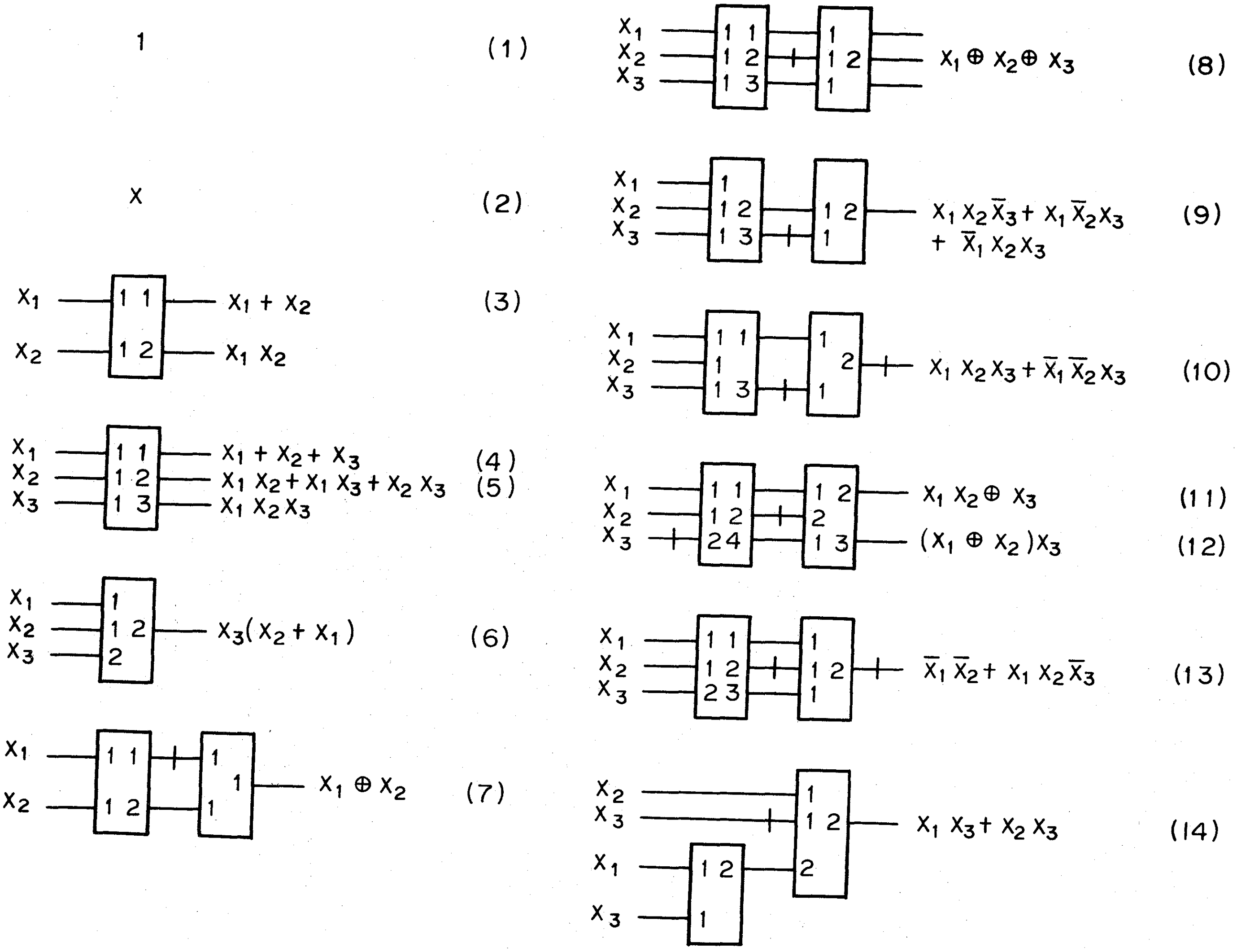

1 (1)
X (2)
X1 X2 11 12 X1 + X2 X1 X2 (3)
X1 X2 X3 11 12 13 X1 + X2 + X3 (4) X1 X2 + X1 X3 + X2 X3 (5) X1 X2 X3
X1 X2 X3 1 12 2 X3(X2 + X1) (6)
X1 X2 11 12 1 1 1 X1 ⊕ X2 (7)
X1 X2 X3 11 12 13 1 12 1 X1 ⊕ X2 ⊕ X3 (8)
X1 X2 X3 1 12 13 12 1 X1 X2 X̄3 + X1 X̄2 X3 + X̄1 X2 X3 (9)
X1 X2 X3 11 1 13 1 2 1 X1 X2 X3 + X̄1 X̄2 X3 (10)
X1 X2 X3 11 12 24 12 2 13 X1 X2 ⊕ X3 (11) (X1 ⊕ X2)X3 (12)
X1 X2 X3 11 12 23 1 12 1 X̄1 X̄2 + X1 X2 X̄3 (13)
X2 X3 X1 X3 12 1 1 12 2 X1 X3 + X2 X3 (14)

Figure 6—Synthesis example of 3-variable functions

of the number of SWV to that of the representative boolean functions is very low. So this table look up method is effective. The minimum value of *W* to realize all 4-variable boolean functions by Figure 4 is 12. The minimum value of *W* may be decreased by Figure 5.

Figure 6 presents synthesis examples for all 14 representative functions of 3 variables. All of them are realized by at most two MMLs, each of which has at most 4 as the weight sum, and at most 3 input-pairs. Table II presents those for all 222 representative functions of 4 variables. All of them are realized by at most two MMLs, each of which has at most 10 as the weight sum, and at most 6 input-pairs. 212 functions out of 222 can be realized at most 8 as weight sum and at most 4 input-pairs.

Example 1; No. 132 function (0, 1, 3, 4, 6, 9, 10) can be realized as shown in Figure 7a.

Example 2; No. 62 function (0, 1, 2, 3, 4, 8) is 1-realizable function, and the fourth column is marked by the symbol "*T*" which stands for threshold function (Figure 7b).

How to look up the table is as follows:

Compute the characteristics vector of the given

TABLE II—Synthesis of 4-Variable Functions

NO	CHAR VECTOR(1)	STANDARD SUM(2)	WI(3)	CONNECTION(4)	T(5)
1	0000000000000000				
2	1000000000000000	0	T	1 1 1 1	1
3	2000100101101111	0 1	T	1 1 1 0	1
4	2001101111011000	0 3	0011	2 2 0 0 1-1	2
5	2011111100000011	0 7	0111	2 0 0 0 1 0-1	2
6	2111100000011110	015	1111	0 0 0 0 1 0 0-1	2
7	3001101111211222	0 1 2	T	1 1 2 2	5
8	3011111102202211	0 1 6	1111	1 1-1 0-2 0 0 0	3
9	3011111122222233	1 2 4	1111	-1 0 0 0 1-1 0 0	3
10	3111100202211112	0 114	1111	1 0 0 0-1-1 0 1	2
11	3111102222011110	0 312	1113	1 0 0 0 0-1 2-2 0 0	4
12	3111102222211332	1 212	1111	-1-1 0 0-1 0 2 0	3
13	4002202222222222	0 1 2 3	T	1 1 0 0	2
14	4011111122222233	0 1 2 4	T	1 1 1 2	2
15	4011211221321322	0 1 2 5	122	1-1 0 1-2 0 0 0 0	3
16	4012212211211233	0 1 2 7	1111	1-1 0 0-2 1 0 0	3
17	4022222202202222	0 1 6 7	122	0 0 0 0 0 1-1 0 0	2
18	4022222222222200	0 3 5 6	1111	-1 0 0 0-2 1-1 0	3
19	4111102222211332	0 1 212	1111	-1-1 0 0 1 0 0 2	3
20	4111122222233334	1 2 4 8	1111	0 0 0 0 1-1 0 0	2
21	4111202121312223	0 1 213	122	0 0 0 1-1-1 0 0 1	2
22	4111222121333221	0 2 5 9	122	0 0 0 1-2 0 1-1 0	3
23	4112201111222332	0 1 215	1111	1 1 0 0-3 0 1 0	3
24	4112221113222312	0 1 611	1122	-1 0-1 0-1 0 0 2 0 0	3
25	4112221133022112	2 3 411	122	0 0 0 1-2 0 0 1 0	2
26	4112221133222334	1 2 411	1111	1-1 0 0 0 2-2 0	5
27	4122211102213321	0 1 615	1122	1 0 0 0 1-2 0 0 1 0	3
28	4122211122233343	1 2 415	1111	-1 0 0 0 1-1 0 2	3
29	4222202022222222	0 11415	122	0 0 0 0-1 0 0 0 1	1
30	4222202222202220	0 31215	1122	0 0 0 0 0 0 1-1 0 0	2
31	4222202222222002	0 31314	1122	0 0 0 0-1 0 0 1-1 0	2
32	5012212233233233	0 1 2 3 4	T	1 1 2 3	5
33	5022222222222244	0 1 2 4 7	1111	-1 0 0 0-1 1 0 0	2
34	5022222222422422	0 1 2 5 6	1111	1-1-1 0 0 2 0 0	4
35	5111122222233334	0 1 2 4 8	T	1 1 1 1	2
36	5111222123332243	0 1 2 4 9	1111	1 2 0 0 2-1-2 0	4
37	5112203333222332	0 1 2 312	1111	1 1 0 0-2 0 0 0	2
38	5112221133222334	0 1 2 411	1111	1-1 0 0 0 1-2 0	4
39	5112221331422332	0 1 2 510	123	-1 0 0 1 0 0 1-2 0 0	4
40	5112221333224132	0 1 3 410	1112	1-1 0 0 1 0-2 0 0	4
41	5112223333244334	1 2 3 4 8	1111	-1-1 0 0 0-1 2 0	3
42	5122211122233343	0 1 2 415	1111	1 0 0 0-2 0 1 0	2
43	5122211322431323	0 1 2 514	1122	1 0-1 0 0 3-2 0 0 0	5
44	5122213322211343	0 1 2 712	1112	1 0 0 0 0 1-2 0 0	3
45	5122213324213321	0 1 3 612	1113	1-1 0 0 0 3-2 0 0 0	5
46	5122213344233343	1 2 3 412	122	1 0 0 1-3 1 0 0 0	3
47	5122233302213323	0 1 6 7 8	122	0 0 0 1-1-1 0 1 0	2
48	5122233322233301	0 3 5 6 8	1111	1 1 1 0 0-2 0 0	3
49	5122233322233345	1 2 4 7 8	1111	-1 0 0 0 2-2 1 0	4
50	5122233322433523	1 2 5 6 8	122	0 0 0 1 1 1-2 0 0	4
51	5222202222222442	0 1 21215	1122	0 0 0 0-1 1-1 0 0 1	2
52	5222202222422224	0 1 21314	1122	0 0 0 0 1-1 0 0 1 0	2
53	5222202424222222	0 1 31214	122	0 0 0 1-1-1 0 0 2	2
54	5222222222244444	1 2 4 815	1111	0 0 0 0 1-1 0 1	2
55	5222222222404222	0 1 61013	1112	1 0 0 0 0-1 2-2 0	4
56	5222222224442244	1 2 4 914	1112	-1-1 0 0 0 3-2 0 0	5
57	5222222244222220	0 3 51012	1111	1 1 0 0 0 1-2 0	4
58	5222222442422442	1 2 51012	*1123	0 1 0 0 0 1 0 0-2 2-2 0	5
59	6022322343343333	0 1 2 3 4 5	T	1 1 2	3
60	6023323333233244	0 1 2 3 4 7	1111	1 1-1 0 0-2 0 0	3
61	6033333322422433	0 1 2 5 6 7	1111	-1 0 0 0 1 0-1 0	3
62	6112223333244334	0 1 2 3 4 8	T	1 1 2 2	3
63	6112323234343243	0 1 2 3 4 9	122	0 1 0 1 0 2-1 0 0	4
64	6113322244233334	0 1 2 3 411	1111	1-1 0 0-1 0 0 1	2
65	6122213344233343	0 1 2 3 412	1111	1 2 0 0-2-1 0 0	3
66	6122233322233345	0 1 2 4 7 8	1111	1 0 0 0 0-1 2 0	2
67	6122233322433523	0 1 2 5 6 8	122	0 0 0 1 0 1-2 0 0	3
68	6122313243334234	0 1 2 3 413	1112	1-1 0 0 2-2 0 0 0	4
69	6122333223332254	0 1 2 4 7 9	122	-1 0 0 1-1 0 2 0 0	3
70	6122333223532432	0 1 2 5 6 9	1111	1-1-2 0 0 3 0 0	5
71	6122333243352232	0 2 3 4 5 9	1112	-1-1 0 0-1 0 2 0 0	3
72	6123312233244343	0 1 2 3 415	1111	1 2 0 0-3 0 1 0	3
73	6123332233222345	0 1 2 4 711	1122	1 0 0 0 1-2 0 1 0 0	3
74	6123332233422523	0 1 2 5 611	122	-1 0 0 1 2 0-1 0 0	4
75	6123332235224323	0 1 3 4 611	123	0 0 0 1-1-1 0 1 0 0	2
76	6133322222233354	0 1 2 4 715	1111	1 0 0 0-1-1 1 0	2
77	6133322224335532	0 1 2 5 615	1112	-1 0 0 0 2-2 1 0 0	4
78	6222222222244444	0 1 2 4 815	1111	0 0 0 0-1 0 1 0	1
79	6222222224442244	0 1 2 4 914	1123	-1-1 0 0 0 3 0-2 0 0 0	5
80	6222222442422442	0 1 2 51012	1124	1-1 0 0 0 3 0-2 0 0 0 0	5
81	6222224444244444	1 2 3 4 812	1122	1 1 0 0 0-3 1 0 0 0	3
82	6222304343323333	0 1 2 31213	122	0 0 0 0-1 0 0 1 0	1
83	6222322123343353	0 1 2 4 915	1113	1 0 0 0 0-1 2 0-2 0	4
84	6222322143323335	0 1 2 41113	1111	1 2 0 0 0 1-3 0	5
85	6222322341523333	0 1 2 51013	123	0 0 0 1 0-1 2-2 0 0	4
86	6222322343321533	0 1 2 51112	1123	1 0 0 0 0-1 0 2 0-2 0	4
87	6222324143343133	0 2 3 4 913	123	-1 0 0 1-2 0 0 2 0 0	3
88	6222324343341331	0 2 3 5 912	1122	-1 0 0 0 1 0-1 0 2 0	3
89	6222344343363333	2 3 4 5 8 9	122	0 0 0 0 1 0-1 0 0	2
90	6223303333233442	0 1 2 31215	1122	0 0 1 1 1-2 0 0 0 0	3
91	6223321133233444	0 1 2 41115	1112	1 0 0 0-1-1 0 2 0	2
92	6223321331433442	0 1 2 51015	*1234	0 0 0 0 0 0 1 0-1 1 0 0 0-1	3
93	6223321333235242	0 1 3 41015	1112	1-1 0 0 0 3-2 0 0	5
94	6223323333211444	0 1 2 71112	1112	-1 0 0 0-1 0 2-1 0	3
95	6223341333213244	0 1 4 71011	1122	-1 0-1 0 0 2 0-1 0 0	4
96	6223341353233222	0 3 4 51011	1112	1-1 0 0 0-2 0 2 0	3
97	6223343333033224	0 3 4 7 811	122	0 0 0 1 0 1 1-2 0	4
98	6223343333233042	0 3 4 7 910	1122	1 0 1 1 0 0-2 0 0 0	3
99	6223343333233446	1 2 4 7 811	*1133	0 0 0 0 1-1 1-1 1-1 0 0	4
100	6233311322442433	0 1 2 51415	*1234	0 0 0 0 0 1-1 1 0 0 0 0-1 0	3
101	6233313322222453	0 1 2 71215	1123	0 0 0 0-1 1-1 0 0 1 0	2
102	6233313322422235	0 1 2 71314	1112	1 0 0 0 1-2 0 1 0	3
103	6233333302224433	0 1 6 7 815	1122	1 0 0 0-1 0-1 0 2 0	2
104	6233333322244411	0 3 5 6 815	1112	0 0 0 0-1 1-1 0 1	2
105	6233333322404233	0 1 6 71013	1223	0 0 0 0-1 0 0 0 1 0-1 0	2
106	6333302222433334	0 1 2131415	1122	0 0 0 0 0-1 0 0 1 0	1
107	6333322222415332	0 1 6101315	1223	0 0 0 0-1 0 1-1 0 0 1 0	2
108	6333322244433336	1 2 4111314	1111	0 0 0 0 0 1-1 0	2
109	7033333344444433	0 1 2 3 4 5 6	T	1 1 1 3	4
110	7122333443354434	0 1 2 3 4 5 8	T	1 2 2 3	5
111	7123332453444343	0 1 2 3 4 510	1111	1-1 2 0-2 0 0 0	3
112	7123334433244345	0 1 2 3 4 7 8	1111	1 1-1 0 0-3 0 0	3
113	7123334433444523	0 1 2 3 5 6 8	122	0-1 0 1 0-1 2 0 0	3
114	7133322444453334	0 1 2 3 4 514	1111	1 1 2 0-3 0 0 0	3
115	7133324444233354	0 1 2 3 4 712	122	0 0 0 1-1-1 1 0 0	2
116	7133324444433532	0 1 2 3 5 612	1111	1-1 0 0 2-2 0 0	4
117	7133344422433534	0 1 2 5 6 7 8	1111	1-1 0 0 1 0-2 0	4
118	7133344444455534	1 2 3 4 5 6 8	1111	1 0 0 0-2 1 0-1	3
119	7222224444244444	0 1 2 3 4 812	1111	1 1 0 0-1-1 0 0	2
120	7222324343345335	0 1 2 3 4 813	122	0 0 0 1-1 0 1 1 0	2
121	7222324345343353	0 1 2 3 4 912	122	0 1 0 1 0 2-2 0 0	4
122	7222344343363333	0 2 3 4 5 8 9	1111	-1-1-1 0 2 0 0 0	4
123	7223323333255444	0 1 2 3 4 815	1111	1 1 0 0-2 0 1 0	2
124	7223323335453244	0 1 2 3 4 914	1112	1-1 0 0-2 0 2 0 0	3
125	7223323355233444	0 1 2 3 41112	122	-1 0 0 1-1 0 0 0 2	2
126	7223323553433442	0 1 2 3 51012	1112	1 2 0 0 0 1 0-3 0	5
127	7223341353433444	0 1 2 4 51011	122	0 1 0 1 1 0 0-2 0	4
128	7223343333233446	0 1 2 4 7 811	1111	1 1 0 0 0-2 2 0	3
129	7223343333433264	0 1 2 4 7 910	122	0 0 0 1-1 0 2-1 0	3
130	7223343333633442	0 1 2 5 6 910	1111	-1-1 0 0 0 2 0 0	3
131	7223343335235424	0 1 3 4 6 811	123	0 0 0 1 1 0-2 0 1 0	3
132	7223343335435242	0 1 3 4 6 910	1122	-1 0-1 0-1 0 2 0 0 0	3
133	7223343553455444	1 2 3 4 5 810	1123	-1-2 0 0 0-2 0 3 0 0 0	4
134	7233313344244453	0 1 2 3 41215	1124	0 0-1 0 0 2 0 0-2 1 0 0	4
135	7233313344444235	0 1 2 3 41314	1112	-1 0 0 0 0 2-1 0 0	3
136	7233313544442433	0 1 2 3 51214	123	0 0 0 1-2 0 0 1 1 0	2
137	7233333322244455	0 1 2 4 7 815	1111	1 0 0 0-1-1 2 0	2
138	7233333322444633	0 1 2 5 6 815	1123	0 0 0 0-1 0 0 1-1 1 0	2
139	7233333324442255	0 1 2 4 7 914	1123	0 0 0 0-1 1 0-1 0 0 1	2
140	7233333324642433	0 1 2 5 6 914	1113	1-1 0 0 0 3 0-2 0 0	5
141	7233333344422633	0 1 2 5 61112	1112	-1 0 0 0 0-1 2-2 0	4
142	7233333522442435	0 1 2 5 7 814	1122	1 0 1 0 0 0 1-3 0 0	4
143	7233333524444413	0 1 3 5 6 814	122	-1 0 0 1-2 0 2 0 0	3
144	7233333542422453	0 1 2 5 71012	1123	-1 0 0-1 0-2 0 2 0 0 0	3
145	7233333544424431	0 1 3 5 61012	1122	-1 0 0 0-1 0 2-1 0 0	3
146	7233333544464435	1 2 3 4 5 814	1223	0 0 0 0 1-1 0 0 1 0 0-2	4
147	7233333564444453	1 2 3 4 51012	122	-1 0 0 1 2 0 0-2 0	5
148	7233335524224433	0 1 3 6 7 812	1123	-1 0 0 0 2 0-2 1 0 0 0	4
149	7233335544244455	1 2 3 4 7 812	1112	1 1 0 0 0-3 1 0 0	3
150	7233335544444633	1 2 3 5 6 812	122	-1 0 0 1 3 0-2 0 0	5
151	7333304444433334	0 1 2 3121314	122	0 0 0 1-1-1 0 2 0	2
152	7333322224453354	0 1 2 4 91415	1123	0 0 0 0 0 1-1 1 0-2 0	4
153	7333322244433336	0 1 2 4111314	1111	1 0 0 0 0 1-2 0	3
154	7333322442433552	0 1 2 5101215	1223	-1 1 0 0 0 0 3 0-2 0 0 0	5
155	7333322442633334	0 1 2 5101314	1123	1 0 0 0 0-1 2 0-2 0 0	4
156	7333322444431534	0 1 2 5111214	*1234	0 0 0 0 0 0 1-1 1 0-1 1-1 0	4
157	7333322644433332	0 1 3 5101214	1112	-1 0 0 0 0-1 0 2 0	2
158	7333324444255554	1 2 3 4 81215	1122	0 0 0 0-1 1-1 0 1 0	2
159	7333324444415332	0 1 3 6101213	1122	1 0 0 0 1 0-2 0 1 0	3
160	7333324444455336	1 2 3 4 81314	1122	0 0 0 0 1-1 0 1 0-2	4
161	7333324446453354	.1 2 3 4 91214	1123	-1 0 0 0 0 1-2 0 0 2 0	3
162	7333344444433330	0 3 5 6 91012	1111	0 0 0 0-1 1-1 0	2
163	7333344444433374	1 2 4 7 91012	122	0 0 0 1-1 0 2 0-2	4
164	7333344444633552	1 2 5 6 91012	1122	0 0 0 0 0 1 0-1 0 0	2
165	8044444444444444	0 1 2 3 4 5 6 7	T	1	1
166	8133344444455534	0 1 2 3 4 5 6 8	1111	1 0 0 0-1 0 0-1	2
167	8133444345554443	0 1 2 3 4 5 6 9	1111	1-1 0 0 0-2 0 0	2
168	8134443355444534	0 1 2 3 4 5 611	1111	1-1 0 0-1 0 1 0	2
169	8144433344455543	0 1 2 3 4 5 615	1111	1 0 0 0-2 1 0 0	2
170	8222444444464444	0 1 2 3 4 5 8 9	T	1 1 1	2
171	8223343553455444	0 1 2 3 4 5 810	122	-1 2 0 1-2 0 0 0 0	3
172	8223443454354535	0 1 2 3 4 5 811	123	1 0 0 1-1 0 0 1 0 0	2
173	8224442464444444	0 1 2 3 4 51011	122	1 1 0 0 0-2 0 0 0	2
174	8224444444244446	0 1 2 3 4 7 811	1111	1 1 0 0 0-1 1 0	2
175	8224444444444264	0 1 2 3 4 7 910	122	1 1 0 1 0 0-2 0 0	3
176	8233333544464435	0 1 2 3 4 5 814	122	1 0 0 1-2 0 2 0 0	3
177	8233333564444453	0 1 2 3 4 51012	122	0 0 0 1 0-2 1 0 0	2
178	8233433443365544	0 1 2 3 4 5 815	1112	1 1 0 0-3 0 2 0 0	3
179	8233433463545344	0 1 2 3 4 51013	1112	1-1 0 0-1 0 0 2 0	2
180	8233435245545344	0 1 2 3 4 6 913	122	0 1 0 1-2 0 0 1 0	2
181	8233435443345346	0 1 2 3 4 7 813	1223	1 1 0 0 0 0-2 0 2 0 0 0	3
182	8233435443545524	0 1 2 3 5 6 813	1123	1-1 0 0 0-2 0 2 0 0 0	3
183	8234432453455453	0 1 2 3 4 51015	1122	1 0-1 0-2 0 2 0 0 0	3
184	8234434433255455	0 1 2 3 4 7 815	122	1 0 0 1-2 0 0 1 0	2
185	8234434433455633	0 1 2 3 5 6 815	1112	-1 0 0-1 0-2 2 0 0	3
186	8244422444464444	0 1 2 3 4 51415	122	1 0 0 0-2 1 0 0 0	2
187	8244424444244464	0 1 2 3 4 71215	122	-1 0 0 1 0-2 2 0 0	3
188	8244424444444246	0 1 2 3 4 71314	1112	1 0 0 0 1-2 1 0 0	3
189	8244444422444644	0 1 2 5 6 7 815	1112	1 0 0 0-1 0-1 2 0	2
190	8244444444422644	0 1 2 5 6 71112	1112	0 0 0 0 1 0-1 0 1	2
191	8333324444255554	0 1 2 3 4 81215	1122	1 1 0 0-2 0 0 2 0 0	2
192	8333324444455336	0 1 2 3 4 81314	1111	1 1 0 0 1-2 0 0	3
193	8333324446453354	0 1 2 3 4 91214	1123	-1 0 1 0 0-2 0 2 0 0 0	3
194	8333344444433374	0 1 2 4 7 91012	122	0 0 0 1 1 0-2 0 1	3
195	8333424343356445	0 1 2 3 4 81315	122	0 0 0 1-2 0 1 1 0	2
196	8333424345354463	0 1 2 3 4 91215	1223	-1-1 0 0 0-2 0 2 0 0 0 0	3
197	8333424365334445	0 1 2 3 4111213	1122	-1 0-1 0 0-1 0 0 2 0	2
198	8333444145534445	0 1 2 4 6 91113	1112	1 0 0 1 0-2 0 0 0	2
199	8333444343334447	0 1 2 4 7 81113	1111	1 0 0 0-1 1-2 0	3
200	8333444343374443	0 2 3 4 5 8 915	1113	0 0 0 0-1 1-1 0 1 0	2
201	8333444343534265	0 1 2 4 7 91013	1223	0 1 0 0 0 0 1-2 0 2 0 0	3
202	8334423335464354	0 1 2 3 4 91415	1223	0 1 0 0 0-2 0 0 2-1 0 0	3
203	8334423355244554	0 1 2 3 4111215	1122	1-1 0 0 0-2 0 0 2 0	2
204	8334423355444336	0 1 2 3 4111314	1111	1-1 0 0 0-2 2 0	3
205	8334441353444554	0 1 2 4 5101115	123	-1 0 0 1 0-3 0 2 0 0	3
206	8334443333244556	0 1 2 4 7 81115	1111	-1-1 0 0 0-1 3 0	3
207	8334443333444374	0 1 2 4 7 91015	122	0 0 0 1-2 0 2-1 0	3
208	8334443335246534	0 1 3 4 6 81115	*1234	0 0 0 0 0 1 0 0-1 1-1 0 0 1	3
209	8334443335442356	0 1 2 4 7 91114	*1234	0 0 0 0 1 0 0-1 0 1-1 1 0-1	4
210	8334443355426334	0 1 3 4 6101113	1123	-1 0 0 0 0-2 0 2-1 0 0	3
211	8344413344455345	0 1 2 3 4131415	1123	0 0 0 0 0-1 0 0 1 0 0	1
212	8344433324453365	0 1 2 4 7 91415	1122	-1 0 0 0 0 1-2 0 2 0	3
213	8344433324653543	0 1 2 5 6 91415	*1123	0 1 0 0 1 0-2 0 2-1 0	4
214	8344433344433347	0 1 2 4 7111314	1111	1 0 0 0 1-2 1 0	3
215	8344433344433743	0 1 2 5 6111215	1223	0 0 0 0 1 0 0-1 0 0 1 0	2
216	8444404444444444	0 1 2 312131415	122	0 0 0 0 0-1 0 1 0	1
217	8444422244444446	0 1 2 411131415	1112	0 0 0 0 0-1 0 1 0	1
218	8444422442644444	0 1 2 510131415	*1234	0 0 0 0 0 1-1 1 0 0-1 1-1 0	4
219	8444422444442644	0 1 2 511121415	*1234	0 0 0 0 0 1 0 0-1 0 1-1 0 1	3
220	8444424444422446	0 1 2 711121314	1122	0 0 0 0 1 0-1 0 1 0	2
221	8444444444404444	0 1 6 710111213	122	0 0 0 0 1 0-1 0 1	2
222	8444444444444440	0 3 5 6 9101215	1111	0 0 0 0-1 1-1 1	2

Note: (1) The characteristic vector
(2) Decimal numbers which corresponds to binary input vectors which make f = 1.
(3) The weight vectors of the first stage MML.
(4) How to connect two MMLs.
(5) The threshold of the second MML.

TABLE III—Realization of Boolean Functions by AETL

Number of AETL	Con-nection	Comment	Weight Sum	Number of Input Pairs
1	Fig. 3	Complete set of Median Functions		
2	Simple Cascade Fig. 4	a. Arbitrary Boolean Functions	$W \leqq 2^n - 1$	$\leqq n$
		b. Symmetric n-Variable Functions	$W = n$	$\leqq n$
		c. Partially Symmetric n-Variable Functions. Symmetric Pattern is $(n_1, n_2, \ldots, n_h)$	$W = \prod_{j=1}^{h} (n_j + 1) - 1$	$\leqq n$
	By-passed Cascade Fig. 5	All 3-Variable Boolean Functions	$W \leqq 4$	$\leqq 3$
		All 4-Variable Boolean Functions	$W \leqq 10$	$\leqq 6$
		212 out of All 222 4-Var. Rep. Functions	$W \leqq 8$	$\leqq 4$

function f

$$\{p(0),\ p(1),\ p(2),\ p(3),\ p(4),\ p(1,2),\ p(1,3),\ p(1,4),\ p(2,3),\ p(2,4),\ p(3,4),\ p(1,2,3),\ p(1,2,4),\ p(1,3,4),\ p(2,3,4),\ p(1,2,3,4)\}$$

and look up the table.

The characteristic vector is essentially equivalent to the "invariant,"[11] which corresponds to the representative function uniquely. The relation between them is

$$w(f \oplus x_{i1} \oplus x_{i2} \oplus \cdots \oplus x_{ik}) = 2^{n-1} + w(f) - 2p(i_1, i_2, \ldots, i_k)$$

where

$$w(f) \equiv \sum_{\vec{X} \in V^n} f(\vec{X})$$

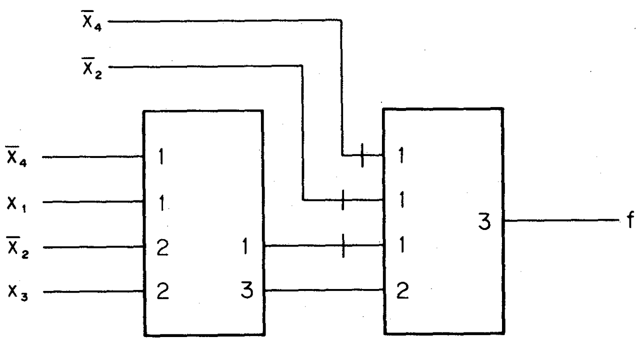
and $f(\vec{X})$ is a boolean function.
Above results are summarized into Table III.

HARDWARE OF AETL[12]

AETL circuit presented in this paper (Figure 8) has following characteristics compared with the usual threshold circuits.[13]

- Since each input-pair is a current switch pair, each threshold is variable and can be given independently.
- Complementary output pair is given and each output can be used as an input for the next stage.
- Weight vector $\vec{W} = (w_1, w_2, w_3, w_4)$ can be selected arbitrarily where $w_1 = w_2 = 1$, $w_3 = 1, 2$, $w_4 = 1, 2, 3, 4$.
- Most of logical functions of four variables can be realized by one or two 4-2-1-1 AETL.
- Since weight sum is large and unit signal amplitude is small, temperature compensation must be secured. So, temperature compensation circuit has been built-in.

CAD has been used to design and simulate the circuit. The influence of accuracy of components and deviations of transistor characteristics have been investigated. Sensitivity factors for circuit parameters have been calculated. The worst case conditions and the probability distributions of circuit component parameters



No. 132 f = (0, 1, 3, 4, 6, 9, 10)

Figure 7a—Synthesis example

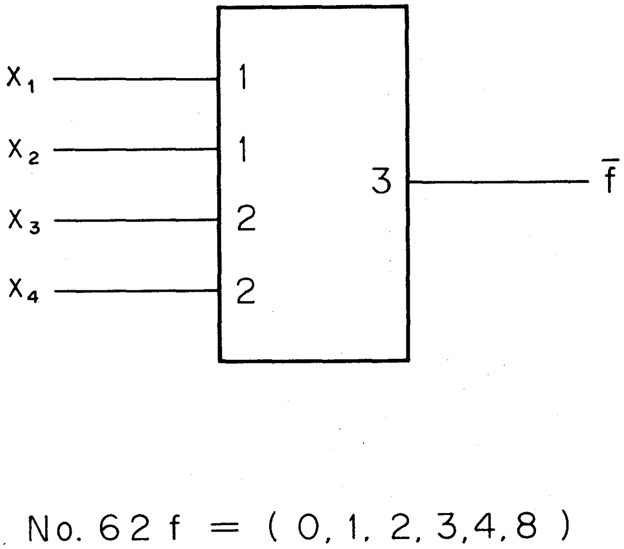


Figure 7b—Synthesis example

have been taken into consideration. Resultant hybrid IC has been successful and used in the plane register described later.

Circuit configuration of AETL

The following two essential operations must be realized.

(1) generation of unit threshold function
(2) weighted analogue summation.

As for (1), the current switch pair is used, which compares the input voltage pair, x_{Uj} and x_{Lj}, then switches the current from the current source according to the result of the comparison. When $(x_{Uj}>x_{Lj})=1$ $(=0)$, *QJU* is turned on (off). As to (2), the current source generates the current proportional to the weight w_j for each pair of inputs. The current from the source flows through *QJU* or *QJL* by the switching operation and is summed at the collector-circuit common resistance *RJ*, and generates the voltage proportional to the sum of each current. As the collector internal resistance of transistor *QJ* is sufficiently high compared with *RJ*, the analogue summation of the currents is achieved almost completely. Unsaturated current switch pair enables high speed logic operation.

Figure 8 shows 4-input-pair AETL circuit $(w_1=w_2=1, w_3=1, 2, w_4=1, 2, 3, 4)$. *QE*1, *QE*2, . . . , *QE*4 form the constant current generators. Diodes *D*1, *D*2, *D*3 compensate the variation of the unit current due to the ambient temperature. Connecting or not the terminals Z_{31}, Z_{41}, Z_{42} to *E* to select the values of emitter resistance *RE*3, *RE*4, the weights W_3, W_4 can be chosen for the logic function to be realized. The voltage level shift circuit by *QH* and *RH*1 is to prevent collector saturation of *QJU* and *QJL*. The minimum value (when $S=8$) of

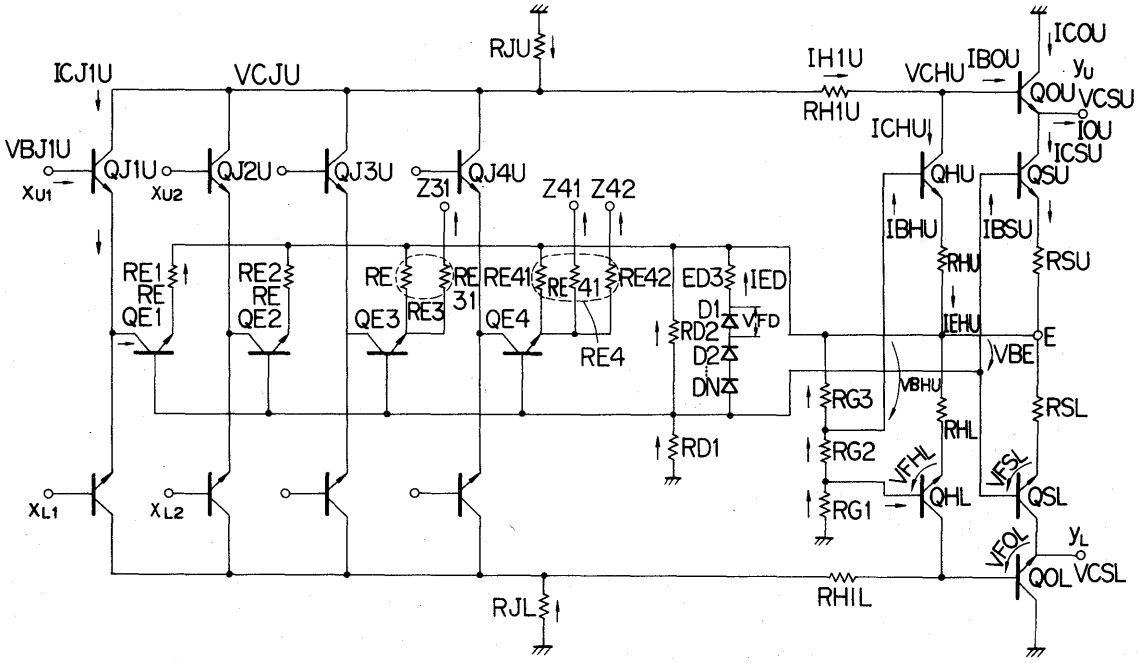
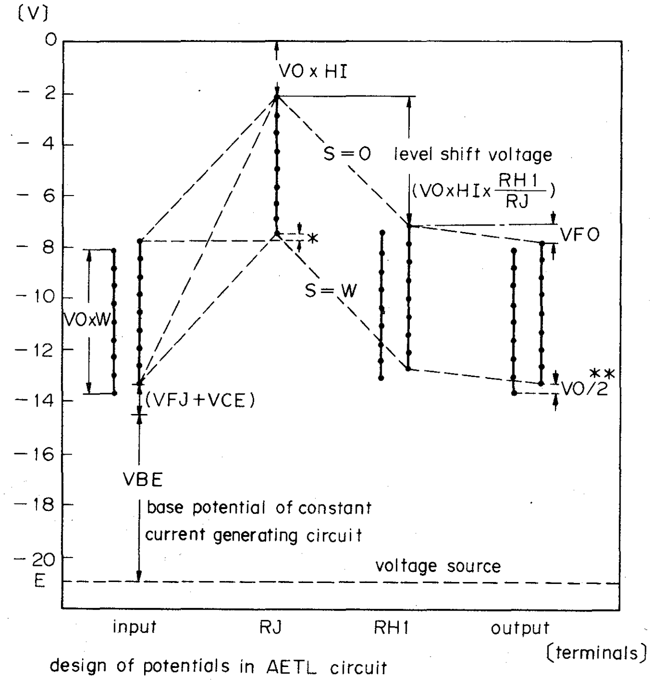


Figure 8—4-2-1-1 AETL circuit



design of potentials in AETL circuit

* tolerance for unsaturation

** level shift for asymmetric outputs

Figure 9—Signal voltage level in 4-2-1-1 AETL

the potential VCJ of the collector of QJ should not be much less than the maximum value (when $S=0$) of output voltage VCS where

$$S=\sum_{j=1}^{4} w_j(x_{Uj}>x_{Lj}) \quad \text{for the upper side.}$$

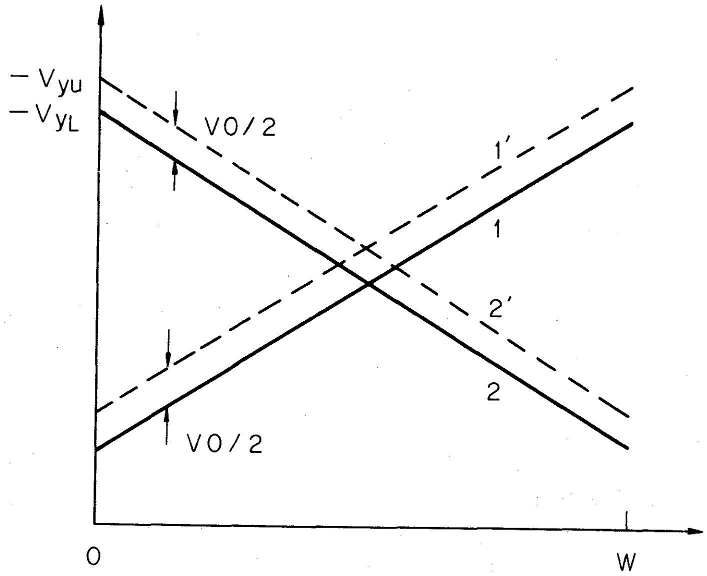
The transition of signal level in the circuit is shown in Figure 9. This circuit is perfectly symmetric in upper and lower sides except the level shift circuit. There is a complementary relation between

$$y_L\equiv \sum_{j=1}^{4} w_j(x_{Uj}>x_{Lj}) \text{ and}$$

$$y_U\equiv \sum_{j=1}^{4} w_j(x_{Lj}>x_{Uj}):$$

$$y_U+y_L=W.$$

For the complete operation of the current switch pair, the situation $x_{Uj}=x_{Lj}$ may not occur. So let two classes of outputs, y and y' be generated, where $y=0, 1, \ldots, W$ and $y'=\frac{1}{2}, 1\frac{1}{2}, \ldots, W+\frac{1}{2}$ (Figure 10). To realize this, the potential of QHU is made a little different from that of QHL. The base emitter voltage VFO of transistor QO influences output voltage directly.



$$y_u = \sum w_i(x_i > x_i'),\; y_L = \sum w_i(x_i' > x_i)$$

Figure 10—Output voltage vs. logical output

QH compensates for this change of VFO caused by temperature variation. In other words, the changes of VFO and VFH are required to be the same for the temperature variation, and resistors to satisfy the relationship of $RJ+RH1 \doteqdot RH$.

4-input-pair ($W_{\max}=8$) AETL circuit was designed and hybrid-integrated (Figure 11). CAD enabled us to design the circuit with appropriate noise margin. The

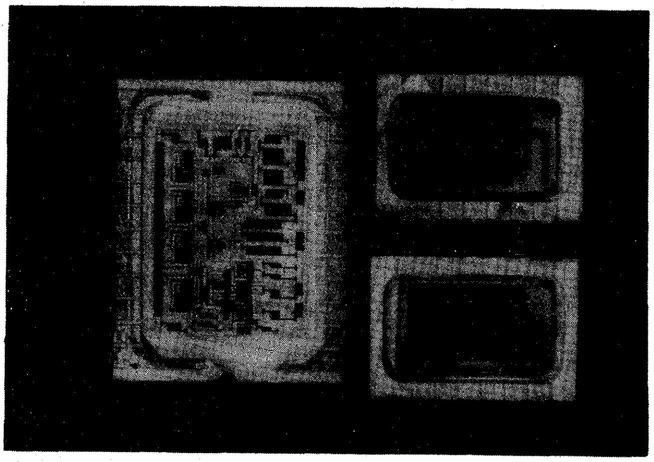


Figure 11—Elements
top; 3-Compatible Switch
left; 4-3-2-1 AETL
bottom; input MML

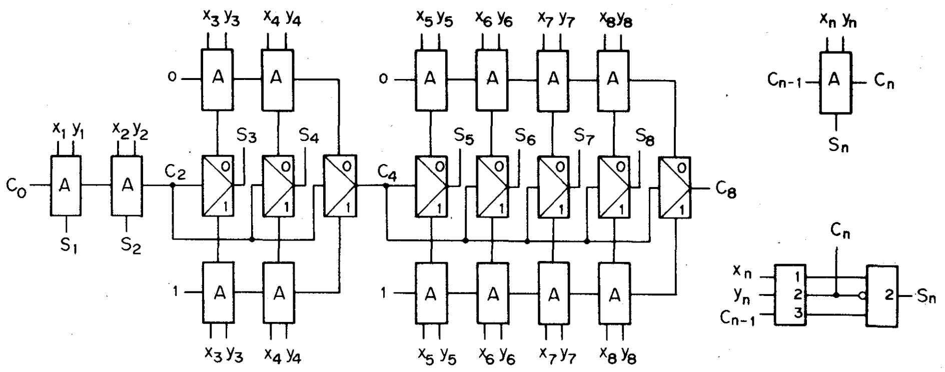


Figure 12—8-bit high speed full adder

results of experiments on integrated AETL agreed very well with that of simulation and the values aimed by the design.

The effect of the ambient temperature was compensated almost completely by the temperature compensation circuit (0.09mV/deg). It was found that the output error was mainly caused by the deviation of resistors. To make $W_{\max}$ large, it is required (1) to increase base emitter break down voltage of transistor *QJ*, (2) to increase the accuracy of resistors.

HIGH SPEED FULL ADDER[14,15,16,17]

To examine the dynamic behavior of hybrid integrated MML we have constructed an 8-bit Full Adder, shown in Figure 12, based on a revised Sklansky's Conditional Sum Method. The experimental results are as follows. The carry propagation time was measured as 11*ns* (Figure 13). Stable operation was assured by a dynamic tester,[18,19] under the conditions that ambient temperature is between $-15°C \sim +75°C$, and the source voltage drops down to 55 percent of the design center value.

PLANE REGISTER

We show, as an example of sequential circuit using hybrid IC AETL mentioned above, a plane register, in which one cell corresponds to one AETL element.

Two types of Data *F/F*'s are shown in Figure 14. They accepts two data inputs D_1, D_2 and are dual each other. The data to be received is decided by the values of two enable signals E_1, E_2. The weight of each input pair is 1. In the figure, integer values denote the logical values of each variable. This plain register operates essentially in multi-valued logic but if we dare to explain it like 2-valued logic, the right side of the virgule corresponds to the "true" and the left side to the "false."

Let us explain the operation of master *F/F* in Figure 14. When $E_1^M = E_2^M = 5$, the *F/F* keeps its previous state. When $E_1^M = 3$ and $E_2^M \geqq 9$ the data D_1^M is enabled by E_1^M and the output Q_U^M follows the data D_1^M. In the next step, E_1^M and E_2^M should be 5, and the output is held. The data D_2^M is enabled when $E_1^M \geqq 9$ and $E_2^M = 3$. When E_1^M, $E_2^M \geqq 9$, the *F/F* is cleared. The slave *F/F* operates in the same way, i.e., when $E_1^S = E_2^S = 6$, the

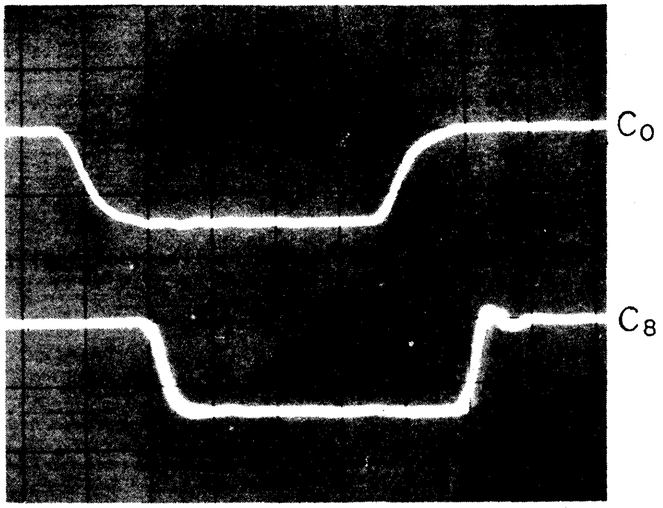


Figure13 —Carry propagation time
C_0; the first bit carry input
c_8; the final bit carry output
X axis; 10ns/div
Y axis; 0.2 V/div
Source voltage; 10V

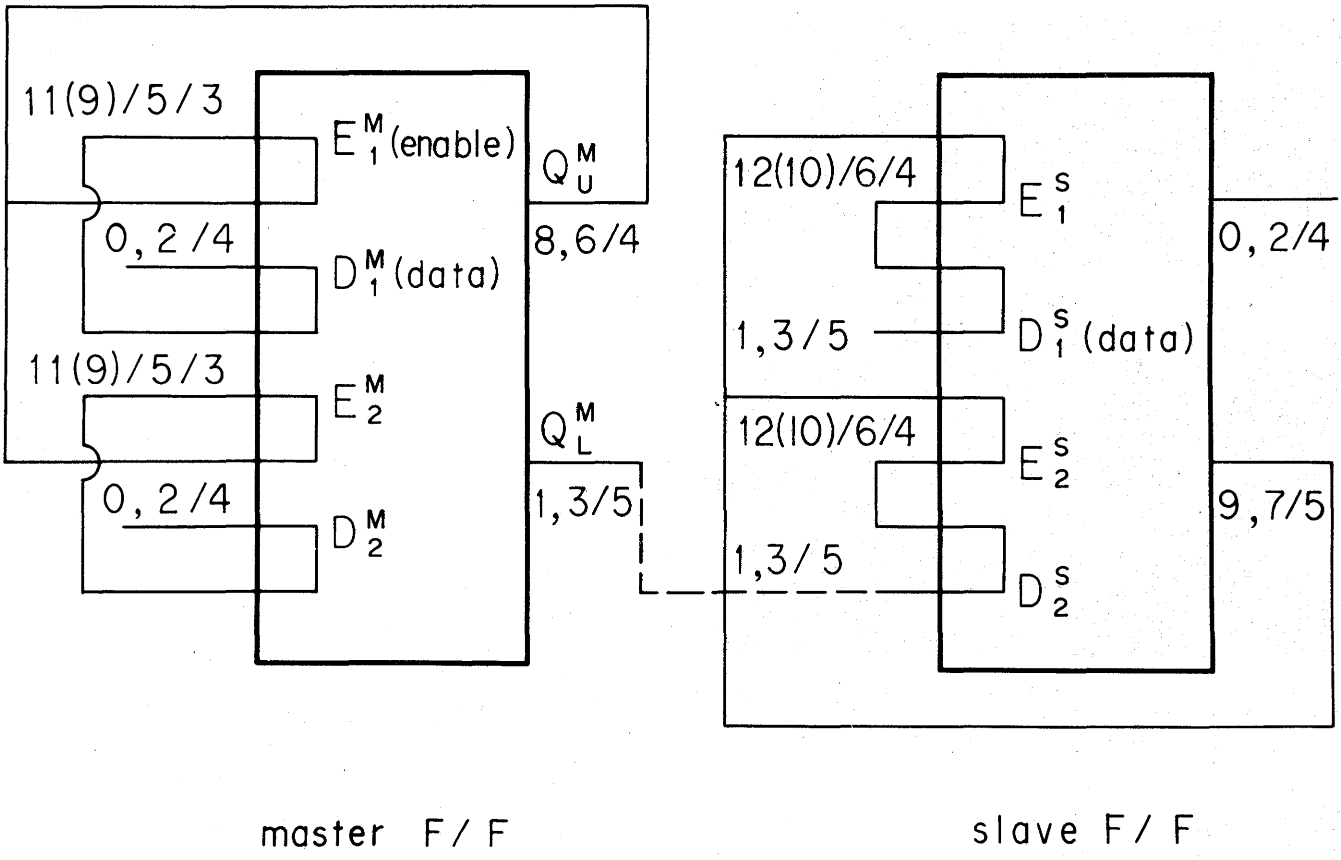
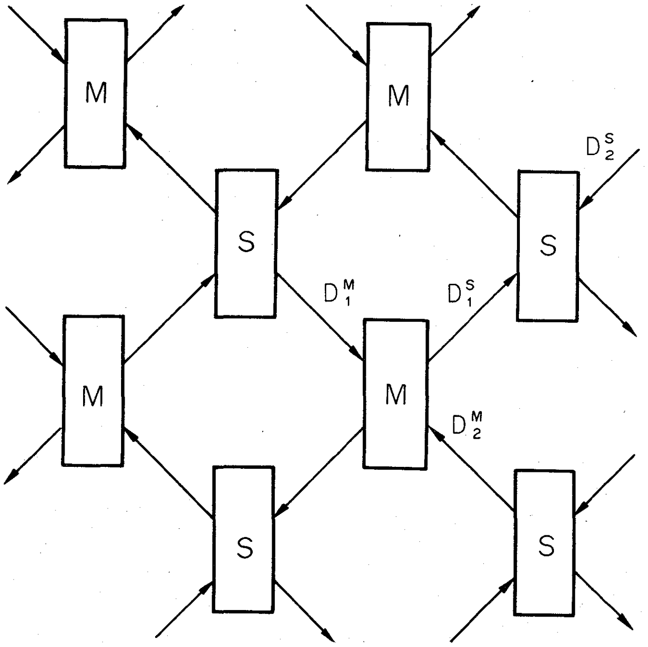


Figure 14— Two data F/F's



Figure—15 Construction of plane register using two data F/F's shown in Figure 14

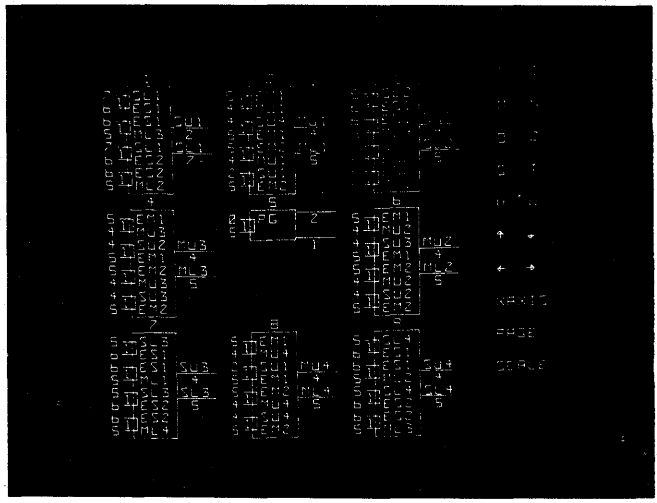


Figure 16—Simulated 4-bit torus register. The terminals which have the same name are connected

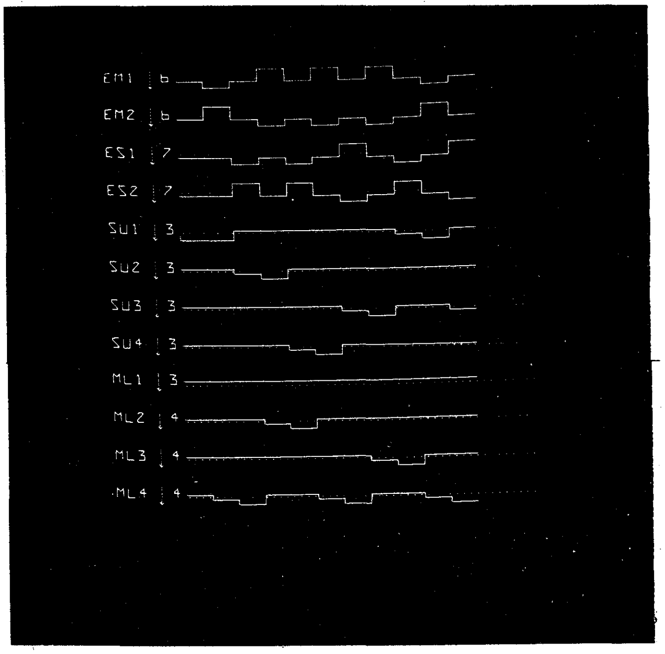


Figure 17—Waveforms of simulated 4-bit torus register. The arrow direction denotes truth value, dot line threshold value between true and false value

F/F keeps its previous state. The data $D_1^S(D_2^S)$ is enabled when $E_1^S=4(\geqq 10)$ and $E_2^S \geqq 10(=4)$. The enable signals can be generated easily by the usual pulse generator and AETL.

The plane register can be made arranging master and slave *F/F*'s above mentioned, as shown in Figure 15.

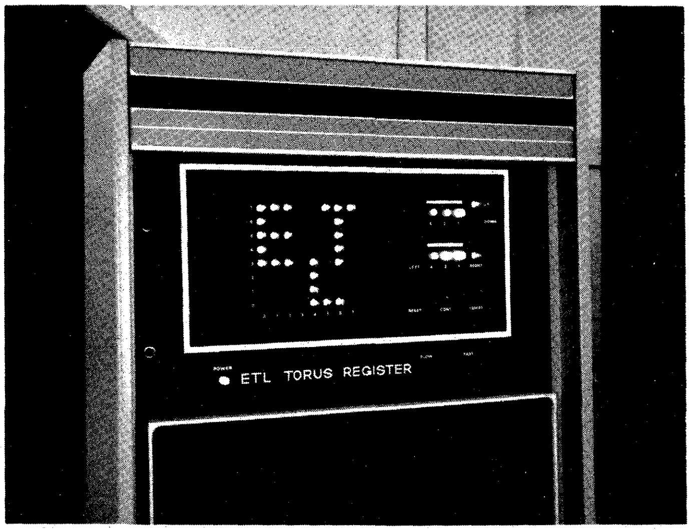


Figure 18—Torus register of 8×8 bits by the hybrid IC AETL circuit

The state of each slave *F/F* on a lattice point can be shifted to any direction by the combination of enable signals. 4 bits plane register connected in the torus form (torus register) was simulated by the AETL simulator (Figure 16). The wave forms of enable signals and outputs are shown in Figure 17 and their values in Table IV. Torus register of 8×8 bits was fabricated using hybrid IC AETL (Figure 18). The stable operation and the ability of AETL for cellular automata were ascertained.

TABLE IV—Simulated Four Bits Torus Register

```
>0 C 4

EM1 EM2 ES1 ES2 SU1 SU2 SU3 SU4 ML1 ML2 ML3 ML4

? 5 ? 5 ? 6 ? 6 * 2 . 4 . 4 . 4 . 5 . 5 . 5 . 5
? 5 ? 5 ? 6 ? 6 + Ø . 4 . 4 . 4 . 5 . 5 . 5 . 5 **

* 3 . 9 ? 6 ? 6 + Ø . 4 . 4 . 4 . 5 . 5 . 5 . 5
* 3 . 9 ? 6 ? 6 + Ø . 4 . 4 . 4 . 5 . 5 . 5 * 3 **

? 5 ? 5 * 4 .1Ø + Ø . 4 . 4 . 4 . 5 . 5 . 5 * 3
? 5 ? 5 * 4 .1Ø . 4 * 2 . 4 . 4 . 5 . 5 . 5 + 1 **

. 9 * 3 ? 6 ? 6 . 4 * 2 . 4 . 4 . 5 . 5 . 5 + 1
. 9 * 3 ? 6 ? 6 . 4 + Ø . 4 . 4 . 5 * 3 . 5 . 5 **

? 5 ? 5 * 4 .1Ø . 4 + Ø . 4 . 4 . 5 * 3 . 5 . 5
? 5 ? 5 * 4 .1Ø . 4 . 4 . 4 * 2 . 5 + 1 . 5 . 5 **

. 9 * 3 ? 6 ? 6 . 4 . 4 . 4 * 2 . 5 + 1 . 5 . 5
. 9 * 3 ? 6 ? 6 . 4 . 4 . 4 + Ø . 5 . 5 . 5 * 3 **

? 5 ? 5 .1Ø * 4 . 4 . 4 . 4 + Ø . 5 . 5 . 5 * 3
? 5 ? 5 .1Ø * 4 . 4 . 4 * 2 . 4 . 5 . 5 . 5 + 1 **

. 9 * 3 ? 6 ? 6 . 4 . 4 * 2 . 4 . 5 . 5 . 5 + 1
. 9 * 3 ? 6 ? 6 . 4 . 4 + Ø . 4 . 5 . 5 * 3 . 5 **

? 5 ? 5 * 4 .1Ø . 4 . 4 + Ø . 4 . 5 . 5 * 3 . 5
? 5 ? 5 * 4 .1Ø * 2 . 4 . 4 . 4 . 5 . 5 + 1 . 5 **

* 3 . 9 ? 6 ? 6 * 2 . 4 . 4 . 4 . 5 . 5 + 1 . 5
* 3 . 9 ? 6 ? 6 + Ø . 4 . 4 . 4 . 5 . 5 . 5 * 3 **

? 5 ? 5 .1Ø * 4 + Ø . 4 . 4 . 4 . 5 . 5 . 5 * 3
? 5 ? 5 .1Ø * 4 . 4 . 4 * 2 . 4 . 5 . 5 . 5 + 1 **
```

```
Note:  (1) '**' denotes stable state.
       (2) '+' and '*' denote 'true'.
           '.' and '?' denote 'false' in this case.
```

ANALOGUE MEMORY[20,21]

A new analogue memory element with similar configuration to AETL has been developed, and its basic experiments have been performed successfully. The principle of this system consists in that (1) the direction of the unit current flowing through each current switch is changed over according to the level of input voltage, (2) the product of the current and the common collector

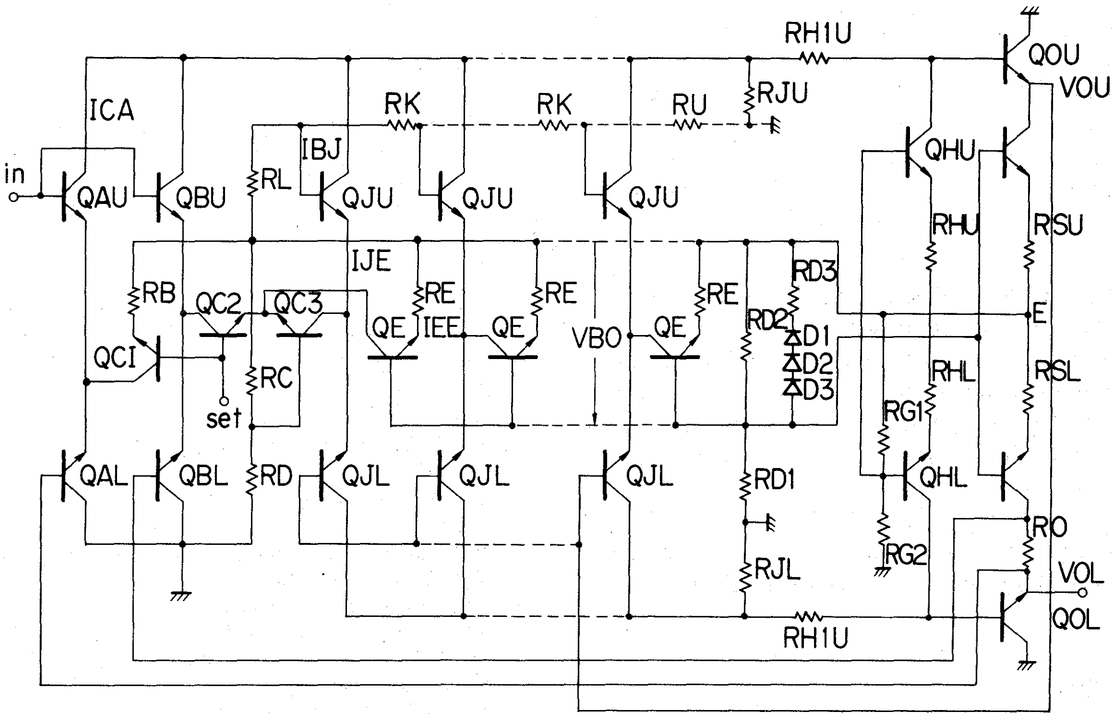


Figure 19—Circuit of analogue memory element

resistance forms an analogue quantity, and (3) the result is made to be self maintained. Results of the basic experiments showed that the response time is μs per step. Figure 19 shows the circuit and Figure 20 shows the input-output response.

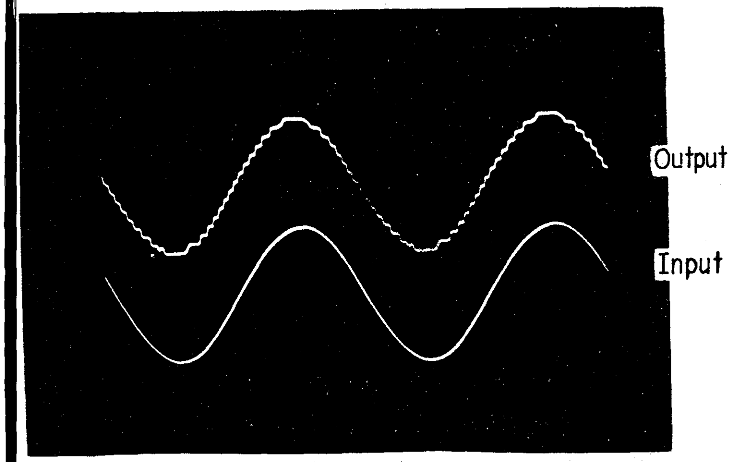


Figure 20—Output response by sine wave input

CONCLUSION

A new kind of threshold logic named AETL (An Extended Threshold Logic) and its simplified modification MML (Multiplex Median Logic) have been presented. This report is summarized as follows:

(1) We obtained several theoretical results for the realization of boolean functions by AETL.

   a. One AETL can generate a complete set of median functions and their negations.
   b. Two AETLs can generate an arbitrary boolean function.
   c. An arbitrary symmetric n-variable boolean function can be generated by two AETLs of which weight sum $W \leqq n$.

   Synthesis examples are shown for all 3-variable (Figure 6) and 4-variable (Table II) boolean functions. All above results can be similarly realized also by MML.

(2) 4-input-pair ($W_{\max}=8$) AETL circuit was designed and hybrid-integrated. It has been used as a unit cell of the plane register.

(3) An 8-bit full adder was fabricated to test the performance of MML.
(4) We showed as an example of sequential circuit using AETL, an 8×8-bit plane register, in which one cell corresponds to one hybrid IC AETL element.
(5) As a first step to realize F/F based on multi-valued logic, 17-valued analogue memory was designed and tested successfully.

Through this research, it is our solid conclusion that the AETL concept will override previous threshold logics in logical abilities. On the other hand, the AETL hardwares presented are by no means conclusive ones. Their future will depend much on IC technology and requirements for the variable weight and variable threshold ability of AETL.

One of our next interest aims at a realization of certain AETL processor, which might present a unified (pre-) processor for various kinds of pattern information.

REFERENCES

1 W S MEISEL
Variable-threshold threshold elements
Doctoral Dissertation E E Dept U of So Calif.
May 1967 and IEEE Transactions pp 656–667 Vol C-17 No 7 July 1968
W S MEISEL
Nets of variable-threshold threshold elements
IEEE Transactions pp 667-676 Vol C-17 No 7 July 1968

2 D R HARING
Multi-threshold threshold elements
IEEE Transactions pp 45–65 Vol EC-15 No 1 February 1966

3 J J AMODEI D HAMPEL T R MAYHEW R O WINDER
An integrated threshold gate
1967 International Solid-State Circuits Conference Digest of Technical Papers pp 114-115 Lewis Winner NY February 1967

4 R O WINDER
The status of threshold logic
1st Annual Princeton Conf on Information Sciences and Systems pp 59-67 Princ Univ NJ March 1967

5 R O WINDER
The status of threshold logic
RCA Review pp 62-84 Vol 30 No 1 March 1969

6 N SANECHIKA
Synthesis of logical functions using Multiplex Median Logic
Bulletin of the Electrotechnical Laboratory pp 17-36 Vol 35 Nos 9 & 10 1971

7 R MORI
Unitron (multiplex median logic)
Technical Group on Electronic Computers of Institute of Electronics and Communication Engineers of Japan December 1968

8 R MORI
Multiplex median logic system
1971 Mexico IEEE International Conference on Systems Networks and Computers pp 683-687 January

9 N SANECHIKA M TAJIMA R MORI
Synthesis of logical functions by Unitron
National Convention of the Institute of Electronics and Communication Engineers of Japan p 957 No 1898 August 1970

10 R MORI Y TSUJI N SANECHIKA
Synthesis of logical functions by Unitron
Joint Convention of the Four Electrical and Electronics Institutes of Japan pp 3566-3567 No 3093 March 1969

11 M A HARRISON
Introduction to switching and automata theory
McGraw-Hill Inc pp 162-167 pp 395-407 1965

12 R MORI Y OKADA M TAJIMA S KAO T TOMARU T ABE
4-input-pair variable weight and variable threshold AETL circuit
Bulletin of the Electrotechnical Laboratory pp 99-125 Vol 35 Nos 9 & 10 1971

13 S COHEN R O WINDER
Threshold gate building blocks
IEEE Transactions pp 816-823 Vol C-18 No 9 September 1969

14 Y TSUJI H TAJIMA R MORI
8-bit high speed full adder by Unitron
National Convention of the Institute of Electronics and Communication Engineers of Japan p 960 No 901 August 1970

15 Y TSUJI N SANECHIKA H TAJIMA R MORI
A high speed full adder using Unitron and Switch
Bulletin of the Electrotechnical Laboratory pp 69-82 Vol 35 Nos 9 & 10 1971

16 J SKLANSKY
Conditional-sum addition logic
IRE Transactions pp 226-231 Vol EC-9 No 2 June 1960

17 R MORI Y TSUJI H TAJIMA
Design and trial fabrication of 3-input push-pull Unitron and compatible Switch
Bulletin of the Electrotechnical Laboratory pp 48-68 Vol 35 Nos 9 & 10 1971

18 H TAJIMA Y TSUJI R MORI
The dynamic tester of small scale logic system
National Convention of the Institute of Electronics and Communication Engineers of Japan p 962 No 903 August 1970

19 H TAJIMA Y TSUJI R MORI T ABE
A dynamic tester for small scale logic system
Bulletin of the Electrotechnical Laboratory pp 140-146 Vol 35 Nos 9 & 10 1971

20 R MORI S KAO
Analog memory element using current switches
Bulletin of the Electrotechnical Laboratory pp 132-139 Vol 35 Nos 9 & 10 1971

21 R MORI S KAO
Stabilization of 16-input-pair MML circuit
Bulletin of the Electrotechnical Laboratory pp 126-131 Vol 35 Nos 9 & 10 1971

Multiple operand addition and multiplication

by SHANKER SINGH and RONALD WAXMAN

International Business Machines Corporation
Poughkeepsie, New York

INTRODUCTION

Traditionally, adders used in small- and medium-sized computers are designed to add two n-bit numbers. There are arithmetic operations which require the addition of a large number of numbers. Multiplication (division) and special function generation are such operations. In large computers, "carry save addition", which adds a group of 3 numbers and reduces their sum to a partial sum of two numbers, has been frequently used to speed up multiplication. One of these two partial sums evaluates the sum modulo 2 of bits in the same binary order; the second partial sum being composed from carries generated but not transferred. These partial sums are regrouped in triplets and enter a "carry look ahead" adder to provide the final sum. The circuit implementation is a cascade connection of full adders, and is referred to in the literature as "adder tree".[1,2] The operation time is considerably reduced because carries are not transferred, although they are formed.

This paper considers the problem of adding k n-bit numbers (operands) where $k>3$. A novel scheme for adding k numbers will be described. It will be shown that by partitioning these k numbers columnwise, such that each column partition contains m bits of each of the k numbers, where m is an integer$\geq \log_2 (k-1)$, the final sum can be obtained in $m+1$ addition cycles. These cycles are not algorithmically related to the cycles used in the adder tree method. Cycle time is dependent upon structure and technology.

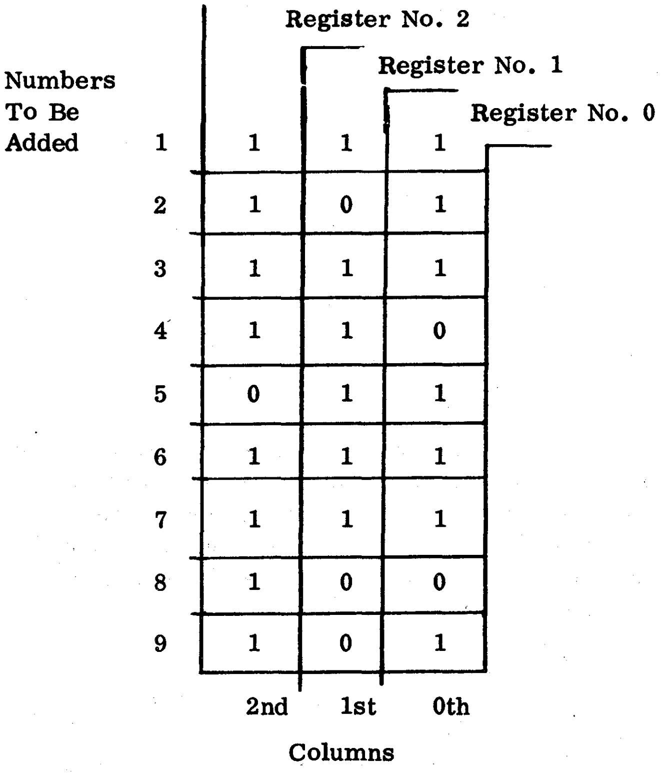
We shall also be describing the use of the adder in multiplication of two numbers. It will be shown that the use of such an adder can lead to a good compromise between hardware requirements and speed for multiplication.

Finally, it will be shown that, from the point of view of large scale integration, such implementation may be quite suitable for arithmetic units in digital computers.

In order to illustrate the basic ideas involved in the method, it will be worthwhile to start with an example. Consider a matrix of nine 3-bit numbers as shown in Figure 1. We can use a circuit Figure 2 to obtain the final sum. The circuit operation can be described step by step as follows:

1. Initialization
   Reset all the register cells $R_1{}^s$ to zero state.
2. 1st Add Cycle
   Gate column 2 (left most) of the matrix in the adder. The sum and the carries appear simultaneously at S_0, $C_0{}^1$, $C_0{}^2$, $C_0{}^3$ terminals of the circuit, which in turn provides the 1st cycle partial sum of the numbers at terminals S_{c2}, S_{c1}, S_{c0}, S_2, S_1, S_0 equal to 001000. The values at S_{c1} down through S_0 are set into register cells $R_1{}^s$ at this time.
3. 2nd Add Cycle
   Gate the column 1 of the matrix in the adder. Once again, the sum and the carries are generated simultaneously and in turn automatically get added to the previous cycle shifted partial sum contained in register cells $R_1{}^s$. (The previous cycle partial sum is effectively shifted left one position because the contents of each register cell is fed into the sub-adder position to its left.) This provides the second cycle partial sum at S_{c2}, S_{c1}, S_{c0}, S_2, S_1, S_0 as 010110.
4. 3rd Add Cycle
   Gate column 0 of the matrix. The operation repeats as in Step 3, and we obtain the final sum 110011 of the 9 numbers in the matrix of Figure 1.

From the example, it may be noted that sub-adder unit 1 of the multiple adder is the most complex and requires the maximum number of logic gates. This subunit also increases in size and complexity as k increases.



Numbers To Be Added	Register No. 2	Register No. 1	Register No. 0
1	1	1	1
2	1	0	1
3	1	1	1
4	1	1	0
5	0	1	1
6	1	1	1
7	1	1	1
8	1	0	0
9	1	0	1
Columns	2nd	1st	0th

Figure 1—Matrix of nine 3-bit numbers stored in three registers of length 9

The size of other sub-adder units remains constant. However, with recent advances in large scale circuit integration and with the availability of monolithic read-only memories, the circuit realization of sub-adder unit 1 should not be difficult. One of the many such possible circuit realizations of sub-adder unit 1 is shown in Figure 3. The decoders 1 and 2 produce all 5-tuples and 4-tuples of inputs a_0, a_1, a_2, a_3, a_4 and a_5, a_6, a_7, a_8 respectively. p_0 represents 5-tuple $\bar{a}_0\bar{a}_1\bar{a}_2\bar{a}_3\bar{a}_4$ (00000). p_1 represents the set union of all the 5-tuples with weight 1 (i.e. {00001+00010+00100+01000+10000}) realized by 'dot ORing' all the tuples of weight 1, (weight is defined as the number of 1^s in the n-tuple). Similarly p_2, p_3, p_4 and p_5 are realized by 'dot ORing' all the 5-tuples of weight 2, 3, 4, and 5 respectively. q_0, q_1, q_2, q_3, and q_4 are also realized by 'dot ORing' all the 4-tuples of weight 0, 1, 2, 3, and 4 respectively. Thus, logical functions for S_0, C_0^1, C_0^2 and C_0^3 can be expressed as fol-

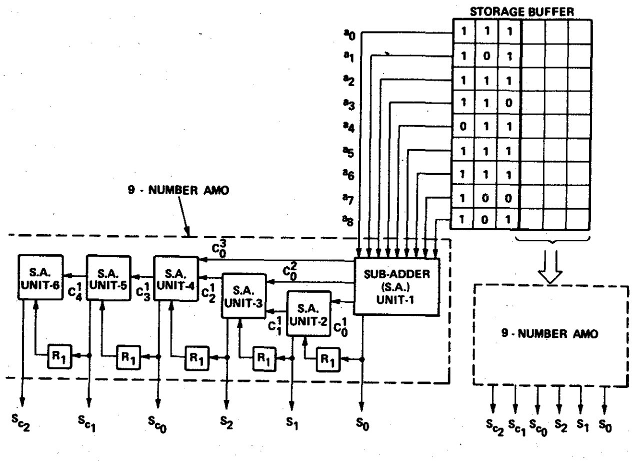


Figure 2—9-number adder

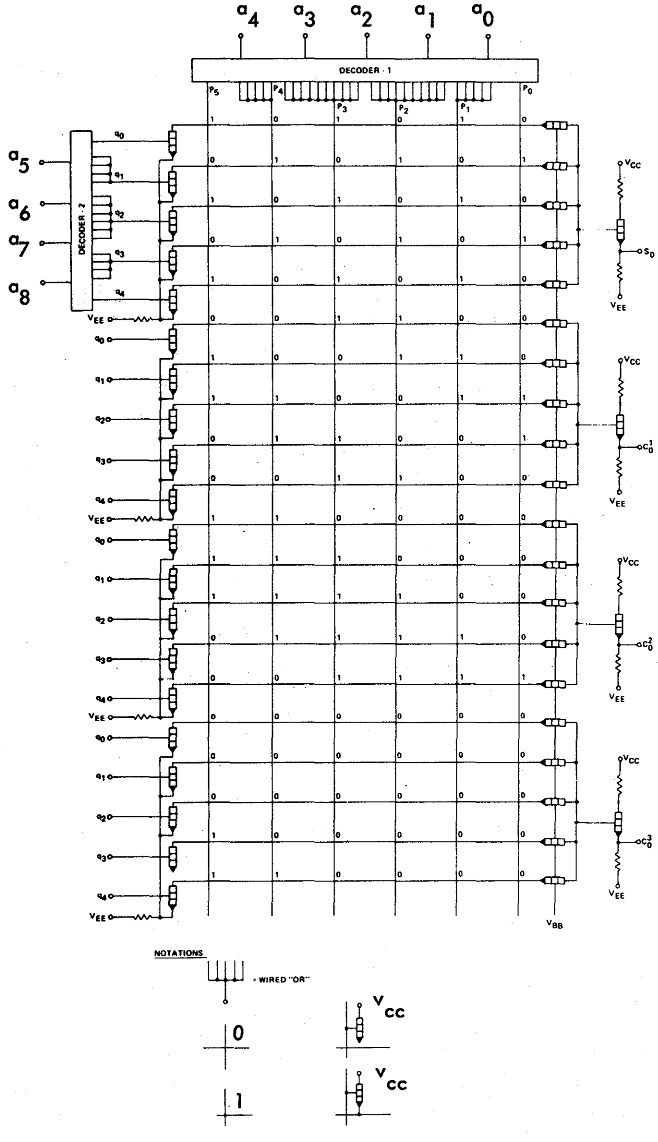


Figure 3—Sub-adder unit 1

lows:

$$S_0=(q_0+q_2+q_4)(p_1+p_3+p_5)+(q_1+q_3)(p_0+p_2+p_4)$$

$$\begin{aligned}C_0^1={}&q_0(p_2+p_3)+q_1(p_1+p_2+p_5)+q_2(p_0+p_1+p_4+p_5)\\&+q_3(p_0+p_3+p_4)+q_4(p_2+p_3)\end{aligned}$$

$$\begin{aligned}C_0^2={}&q_0(p_4+p_5)+q_1(p_3+p_4+p_5)+q_2(p_2+p_3+p_4+p_5)\\&+q_3(p_1+p_2+p_3+p_4)+q_4(p_0+p_1+p_2+p_3)\end{aligned}$$

$$C_0^3=q_3p_5+q_4(p_4+p_5)$$

The other function shown in Figure 2 for sub-adder units 2, 3, 4, 5 and 6 are expressed as:

$$S_1=[S_0 \text{ (previous cycle)}]\oplus C_0^1$$

$$C_1^1=[S_0 \text{ (previous cycle)}]\cdot C_0^1$$

$$S_2=[S_1 \text{ (previous cycle)}]\oplus C_1^1\oplus C_0^2$$

$$C_2^1=[S_1 \text{ (previous cycle)}]\cdot[C_1^1+C_0^2]+C_1^1\cdot C_0^2$$

$$S_{c0}=[S_2 \text{ (previous cycle)}]\oplus C_2^1\oplus C_0^3$$

$$C_3^1=[S_1 \text{ (previous cycle)}]\cdot[C_2^1+C_0^3]+C_2^1\cdot C_0^3$$

$$S_{c1}=[S_{c0} \text{ (previous cycle)}]\oplus C_3^1$$

$$C_4^1=[S_{c0} \text{ (previous cycle)}]\cdot C_3^1$$

$$S_{c2}=[S_{c1} \text{ (previous cycle)}]\oplus C_4^1$$

An adder of the type shown in Figure 2 is able to add any 9 numbers of n bits long, with the final sum available after n cycles. Such an adding scheme has an addition time proportional to n. Therefore, if we use many parallel adder units such as shown in Figure 2, we increase the speed of addition considerably.

The example shown added the most significant column first. However, equivalent results would be obtained if the least significant column were to be added first. This may be verified easily by the reader. One less S.A. unit and one more R_1 unit (see Figure 2) would be required.

Let us proceed with the example, but increase to 32 bits the length of the nine numbers to be added. Suppose we partition these nine numbers column-wise, such that each partition set has three adjacent columns. Now the eleven partition sets, each with nine numbers, are added in parallel, using 11 adder units of Figure 2. (See Figure 3.) The final sum digits from 11 units denoted by S_0, S_1, . . . S_{32} are stored in register A, and the sum digits denoted by S_{c0}, S_{c1}, . . . S_{c32} are stored in register B. Three cycles of AMO are required to obtain the register A and B sums. In the fourth cycle, the contents of registers A and B are fed to a carry look-ahead adder to obtain the final sum of all nine 32-bit numbers.

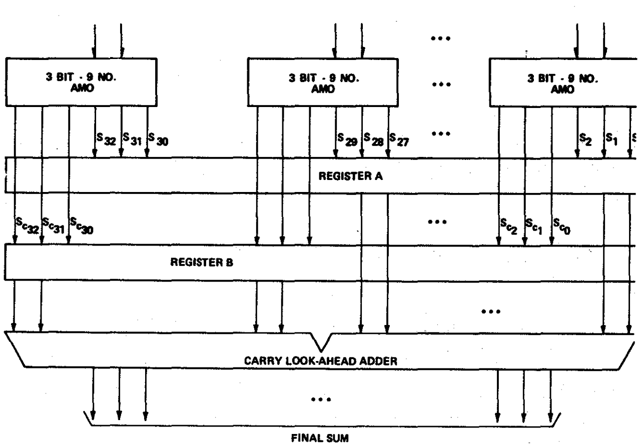


Figure 4—32-bit 9-number adder

Thus in 4 addition cycles, one may add nine numbers. Note that the register positions are lined up into the carry look-ahead adder so that S_{32} adds to S_{c29}, S_{31} to S_{c28}, S_{30} to S_{c29}, etc. Thus a 36 position CLA is required.

The first three cycles include the time to ripple the carriers through sub-adder unit 2 to 6. But this is the case for a simple design. In a more sophisticated design using techniques of "carry look ahead", one could reduce each individual cycle time for the first three cycles to a minimum by generating carriers C_1^1, C_2^1, C_3^1, and C_4^1, simultaneously. Such expressions of carry generation are given by:

$$C_1^1=C_0^1[S_0 \text{ (previous cycle)}]$$

$$\begin{aligned}C_2^1={}&C_0^1[S_0 \text{ (previous cycle)}][S_1 \text{ (previous cycle)}+C_0^2]\\&+[S_1 \text{ (previous cycle)}]C_0^2\end{aligned}$$

$$\begin{aligned}C_3^1={}&C_0^1[S_0 \text{ (previous cycle)}][S_1 \text{ (previous cycle)}+C_0^2]\\&\times[S_2 \text{ (previous cycle)}+C_0^3]+[S_2 \text{ (previous cycle)}]C_0^3\end{aligned}$$

$$\begin{aligned}C_4^1={}&[S_{c0} \text{ (previous cycle)}][C_0^1[S_0 \text{ (previous cycle)}]\\&\times[S_1 \text{ (previous cycle)}+C_0^2][S_2 \text{ (previous cycle)}\\&+C_0^3]+[S_2 \text{ (previous cycle)}C_0^3]]\end{aligned}$$

The circuit implementation of these expressions to obtain the sum term will lead to a minimum overall time for addition of nine numbers. It can be easily verified that if each column partition of nine numbers had two adjacent columns instead of three, then it would be impossible to obtain two partial sums $000S_{31}S_{30}$ . . . S_0 and S_{c31}, S_{c30}, . . . S_{c0} 000 only by appending S_i and S_{ci} for $f=0$, 1 . . . 31 from individual sub-adder units in Figure 4 respectively. However, one could obtain at least three partial sums which can be formed by appending only. It may be noted that if we intended to add five numbers, a partition set of two adjacent columns would be quite suitable.

The reader may also note that the specific manner in which the partial sum digits are separated for register A and B is no consequence provided the individual digits preserve proper positional significance.

ADDITION OF k OPERANDS

Consider a design for an adder for multiple operands (AMO) capable of adding k, n-bit long numbers in minimum addition cycles by a scheme such as described by the example of nine numbers. First, we shall show that it is possible to add k numbers in $m+1$ cycles, where m is the smallest integer $\geq \log_2 (k-1)$.

Let each column partition of k numbers have m adjacent columns for any k such that $(2^{m-1}+1) < k \leq 2^m+1$. It is easy to show that sum of k m-bit long numbers is always $\leq 2m$ bits for the given range of k. Thus, we can always represent these sum bits by $\{S_{c(m-1)}S_{c(m-2)} \ldots S_{c(0)}, S_{m-1}S_{m-2} \ldots S_0\}$. These sum bits can always be expressed as two partial sums;

$$\{\underbrace{00\ldots0}_{\substack{m\\ \text{bits}}}\ \underbrace{S_{m-1}S_{m-2}\ldots S_0}_{\substack{m\\ \text{bits}}}\}$$

$$\text{and}\ \underbrace{S_{c(m-1)}S_{c(m-2)}\ldots S_{c(0)}}_{\substack{m\\ \text{bits}}}\ \underbrace{00\ldots0}_{\substack{m\\ \text{bits}}}$$

Therefore we can always group the total sum bits from adding n bit numbers with m bits per partition, using n/m m bit adder units, as two partial sums consisting of:

$$\underbrace{00\ldots0}_{\substack{m\\ \text{bits}}}\ S_{m-1}{}^1S_{m-2}{}^1\ldots S_0{}^1S_{m-1}{}^2S_{m-2}{}^2\ldots S_0{}^2\ldots$$

$$S_{m-1}{}^{n/m}S_{m-2}{}^{n/m}\ldots S_0{}^{n/m}$$

and

$$S_{c(m-1)}{}^1S_{c(m-2)}{}^1\ldots S_{c(0)}{}^1S_{c(m-1)}{}^2S_{c(m-2)}{}^2\ldots S_{c(0)}{}^2\ldots$$

$$S_{c(m-1)}{}^{n/m}S_{c(m-2)}{}^{n/m}\ldots S_{c(0)}{}^{n/m}\ \underbrace{00\ldots0}_{\substack{m\\ \text{bits}}}$$

by applying the operation of appending only. (In the previous example of adding nine 32 bit numbers, this notation will yield two partial sums of:

$$0\ 0\ 0\ S_2{}^1\ S_1{}^1\ S_0{}^1\ S_2{}^2\ S_1{}^2\ S_0{}^2\ S_2{}^3\ S_1{}^3\ S_0{}^3\ldots$$

$$S_2{}^{10}S_1{}^{10}S_0{}^{10}S_2{}^{11}S_1{}^{11}S_0{}^{11}$$

$$S_{c(2)}{}^1S_{c(1)}{}^1S_{c(0)}{}^1S_{c(2)}{}^2S_{c(1)}{}^2S_{c(0)}{}^2S_{c(2)}{}^3S_{c(1)}{}^3S_{c(0)}{}^3$$

$$\times S_{c(2)}{}^4S_{c(1)}{}^4S_{c(0)}{}^4\ldots S_{c2}{}^{11}S_{c1}{}^{11}S_{c0}{}^{11}\ 0\ 0\ 0$$

this corresponds to $000S_{32}S_{31}\ldots S_0$ and $S_{c32}S_{c31}\ldots S_{c0}\,000$.)

However, by choosing $m-1$ adjacent columns instead of m, for the same range of k, it will be impossible to obtain only two partial sums. In case of $m-1$, it is easy to show that one cannot obtain less than three partial sums formed by simply appending the sums available from individual 2 bit, 9 number AMO units in Figure 4. Addition of these three partial sums implies one additional stage of carry save addition (i.e., another cycle of addition) before using the carry look ahead adder to obtain the final sum of k numbers. Thus, minimally $m+1$ cycles are required to add k numbers.

The reader may note that according to the general partition concept explained with regards to k number addition, carry save adder design is a special case. For carry save addition, $k=3$ and hence from the minimum addition cycle point of view, each column partition set can have one column only. While for $k=5$, 9, 17 and 33, the column partitions must have minimally 2, 3, 4, and 5 adjacent columns respectively. For example, the addition of 33 numbers with 5 bit column partitions will require only 6 addition cycles.

APPLICATION OF ADDER FOR MULTIPLE OPERANDS (AMO) FOR MULTIPLICATION

Figure 5 illustrates the long hand process of the procedure for the general case of multiplying an n-bit

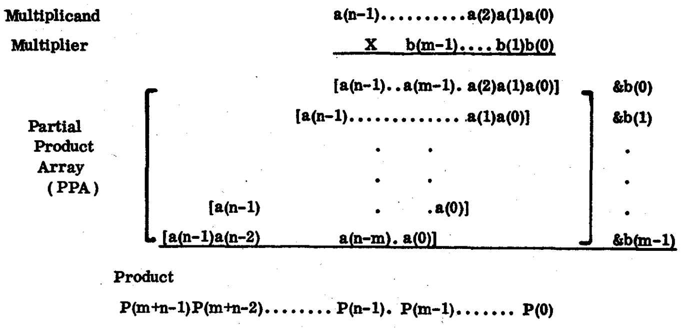

Multiplicand
a(n-1)..........a(2)a(1)a(0)
Multiplier
X b(m-1)....b(1)b(0)
Partial Product Array (PPA)
[a(n-1)..a(m-1). a(2)a(1)a(0)]
&b(0)
[a(n-1).............a(1)a(0)]
&b(1)
[a(n-1)
.a(0)]
[a(n-1)a(n-2)
a(n-m). a(0)]
&b(m-1)
Product
P(m+n-1)P(m+n-2)........P(n-1). P(m-1)....... P(0)

Figure 5—Partial product array (PPA)

```
              a(8)a(7)a(6)a(5)a(4)a(3)a(2)a(1)a(0)
              x         b(5)b(4)b(3)b(2)b(1)b(0)
              a(8)a(7)a(6)a(5)a(4)a(3)a(2)a(1)a(0)        b(0)
          a(8)a(7)a(6)a(5)a(4)a(3)a(2)a(1)a(0)            b(1)
      a(8)a(7)a(6)a(5)a(4)a(3)a(2)a(1)a(0)                b(2)
  a(8)a(7)a(6)a(5)a(4)a(3)a(2)a(1)a(0)                    b(3)
a(8)a(7)a(6)a(5)a(4)a(3)a(2)a(1)a(0)                      b(4)
(8)a(7)a(6)a(5)a(4)a(3)a(2)a(1)a(0)                       b(5)
```

Figure 6a—A 9-bit X 6-bit example

number by an m-bit number. Once the partial product array (PPA) is established, the product is obtained by summing the rows of this array. This is where the AMO can be used in several different ways.

Once again, consider an AMO which is designed to add nine numbers. Thus, each AMO partitioned unit is capable of handling a maximum of nine rows of 3 bits. Since we are adding in parallel one column from each 3-column partition set at a time, three cyles of add are sufficient for each nine rows of any PPA. The two partial sums from AMO's are then fed into a carry look-ahead adder to yield the final result.

Naturally, from the point of view of speed of multiplication, it would be desirable to establish the PPA of nine rows in a parallel operation. This can be done by a circuit such as shown in Figure 6. A skewed array is established where each cell position consists of an AND gate and a shift register cell. The multiplicand is applied at the top of the array and multiplier to the side inputs of the array. Each bit of the multiplicand in ANDed with each bit of the multiplier and the result is stored in the corresponding register position. The skewing of the array accomplishes the appropriate shift of multiplicand simultaneously with the data entry. The PPA in Figure 6 is partitioned into three column sections. The left column of each 3-column section feeds the AMO. At the conclusion of the first cycle, the 3-bit shift registers shift left one position, allowing the data from second column of each 3-bit section to feed the adder. This is done one more time and PPA operation is complete.

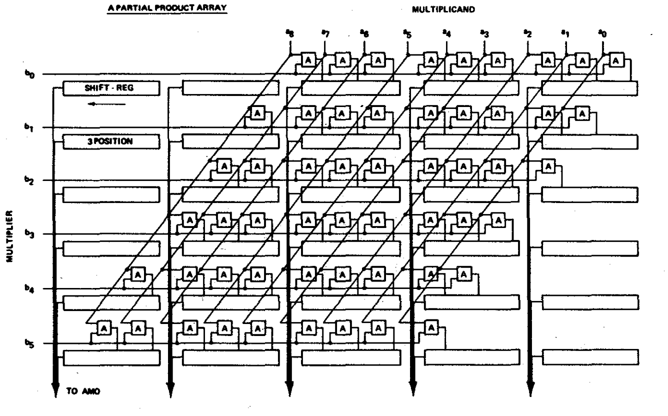


Figure 6b—Circuit realization of PPA

							a(8)	a(7)	a(6)	a(5)	a(4)	a(3)	a(2)	a(1)	a
									x	b(5)	b(4)	b(3)	b(2)	b(1)	b
	0	0	0	0	0	0	a(8)	a(7)	a(6)	a(5)	a(4)	a(3)	a(2)	a(1)	a
	0	0	0	0	0	a(8)	a(7)	a(6)	a(5)	a(4)	a(3)	a(2)	a(1)	a(0)	0
	0	0	0	0	a(8)	a(7)	a(6)	a(5)	a(4)	a(3)	a(2)	a(1)	a(0)	0	0
	0	0	0	a(8)	a(7)	a(6)	a(5)	a(4)	a(3)	a(2)	a(1)	a(0)	0	0	0
	0	0	a(8)	a(7)	a(6)	a(5)	a(4)	a(3)	a(2)	a(1)	a(0)	0	0	0	0
	0	a(8)	a(7)	a(6)	a(5)	a(4)	a(3)	a(2)	a(1)	a(0)	0	0	0	0	0
											0			0	
Column:	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3

Figure 7a—Cyclic nature of PPA

A minimized partial product array

Observation of the array shown in Figure 6 demonstrates the cyclic nature of the PPA. Figure 7 denotes how column 2 and 3 in each three column section may be obtained in the column 1 position. By shifting column 1 up one position, and feeding into the bottom of column 1 the value (previous to the shift) in the third position up of the column 1 immediately to the right, the column 1 position will contain the column 2 values. The values of column 3 may then be obtained in the column 1 position by another upward shift and transfer from the right. Note that with the use of this algorithm, the AND

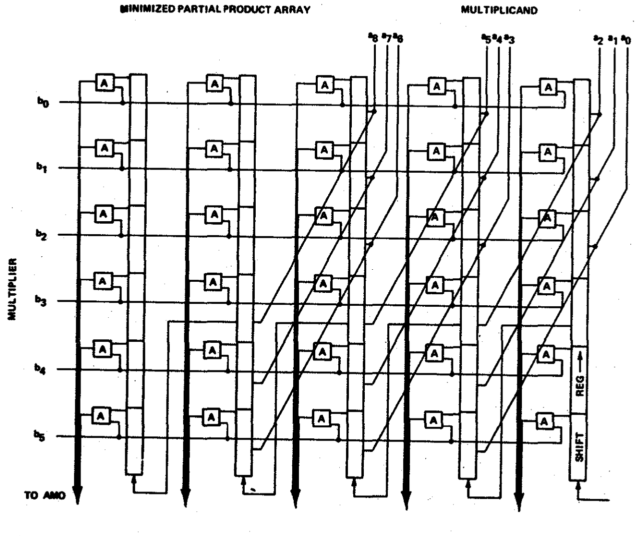


Figure 7b—Circuit realization of PPA with minimum circuits

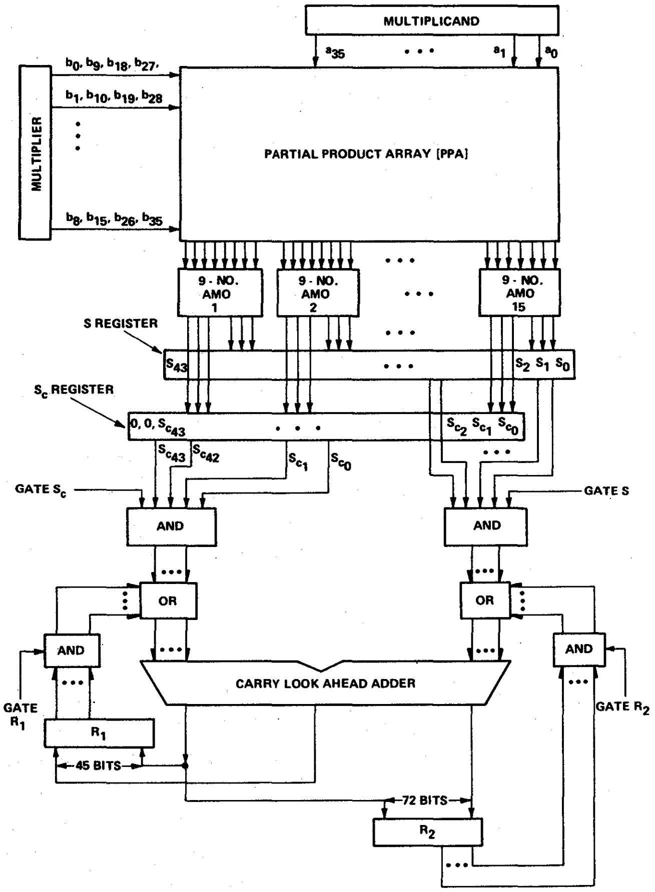


Figure 8—Schematic of 36 × 36 bit multiplier

of the multiplier with the multiplicand takes place at the output of each register position. This technique reduces the matrix to ⅓ of its former size.

Implementation of a 36 bit×36 bit multiplier

The PPA for a 36 bit×36 bit multiplication requires 36 rows and 36 skewed columns or 1296 bits. Including the bit positions containing zeros results in an array of 36 rows by 72 columns or 2592 bits. Without loss of generality, we will consider partitioning the array into one containing nine rows and 44 columns (9×36 skewed). Thus, it will take four passes through the PPA to apply all 36 bits of the multiplier, nine bits at a time, to the multiplicand. We will require 15 3-column partition sets to utilize the three cycle AMO. Also, the minimized PPA will be used. Only 15, 9-bit shift registers with appropriate AND gates are required for this suggested PPA (135 bits of storage).

Four passes through this PPA will take 12 cycles. Figure 8 illustrates the structure of the complete multiplier. The operation is as follows:

1. The multiplicand and the low order nine bits of the multiplier are entered into their respective input registers feeding the PPA.
2. The multiplicand is entered into the PPA, all bits in parallel, for all nine rows.
3. The S and S_c registers are filled in 3 cycles of the multiplier:
   A. On the first cycle, the bits in each column are applied to their respective nine bit adders.
   B. The second cycle starts with the registers all advancing up one position and being fed from the designated position on the right. It concludes with the application of the nine bits of each register to their respective nine bit adders.
   C. The third cycle is a repeat of the second.
4. At this time, the S_c and S registers have a partial result for this pass. The contents of the S and S_c registers are added in the carry look ahead adder. The result is placed in register R_1. R_1 is added to R_2 in the carry look ahead adder with

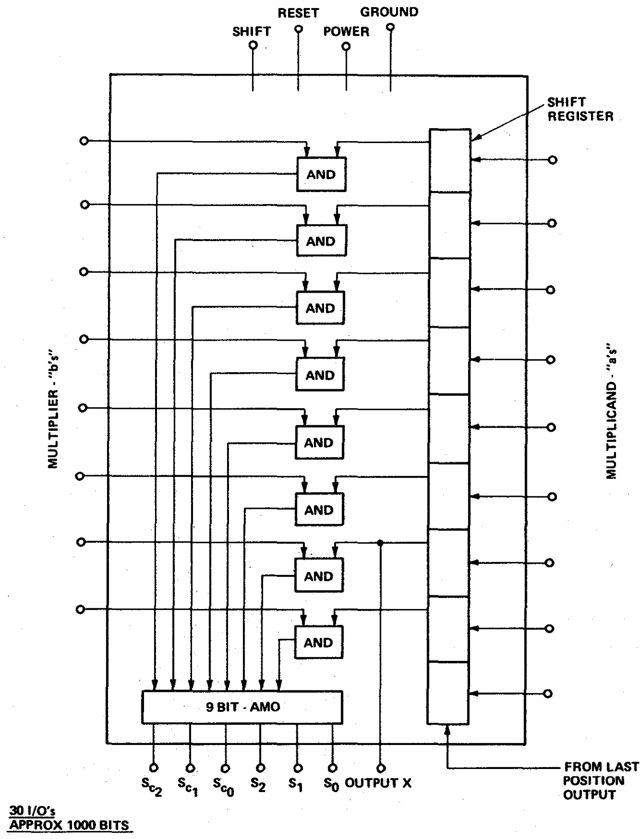


Figure 9—Module partition of a multiplier

the most significant bit of R_2 line up with the ninth bit (to the right of the most significant bit) of R_1, and R_1 left justified as it goes into the adder. The result is placed in R_2, left justified.

5. In parallel (overlapped) with the operations of
6. step 4 is the second pass through the PPA. The
7. multiplicand remains the same, but the second
nine bits of the multiplier are applied to the
indicated multiplier inputs. This is basically a
repeat of steps 1, 2, and 3 in parallel with step 4.
8. Step 4 is repeated for the second pass.
9. Steps 5, 6, and 7 repeated overlapped with step
10. 8. The third nine bits of the multiplier are ap-
11. plied to their inputs.
12. Step 4 is repeated for the third pass.
13. Steps 9, 10, and 11 repeated overlapped with
14. step 12. The fourth nine bits of the multiplier
15. are applied to their inputs.
16. Step 4 is repeated for the fourth pass.
17. The contents of R_2 now is the final product.

If another multiplication is to take place immediately, it can be overlapped with step 16. This results in a 12 cycle multiplication (12 passes through the AMO). The final pass through the carry look ahead adder, when it is not overlapped, adds the equivalent of another 2 to 3 cycles onto the multiply time. Thus, two 36-bit numbers may be multiplied, using this technique and the suggested partitions, in the equivalent of 12 to 15 AMO cycles.

A possible module partition for an n bit×m bit multiplier

Figure 9 illustrates a possible module partition, sliced so that it may be applied to any multiplier size. Nine rows of multiplier may be accommodated at each passs and one module is used for each 3 columns of PPA. Thus, for the 36 bit×36 bit multiplier, 15 modules would be used and four passes required. For a 72×72 bit multiplier, 27 modules would be used and eight passes required. The multiply time may, of course, be halved by approximately doubling the number of modules to handle 18 bits for each pass, using two carry look ahead adders, and going through an extra carry look ahead add cycle with the two partial sums resulting. Thus, for approximately four times the hardware of the 36×36 bit multiplier, a 72×72 bit multiplier could be just as fast.

CONCLUSIONS

If the PPA could be compacted by neglecting all rows containing zeros, then the number of rows in the modified PPA will be the same as the number of 1's in the multiplier. This modified PPA could be further reduced in size by multiplication techniques[3,4] used for shift-over 1's. Thus, in many ways, compromises between overall speed of multiplication, amount of hardware needed and various ways to organize multiplication can be reached. The circuit implementation of a 17-number adder using a Read Only Memory type of circuit seems possible in the near future. This in turn could handle 17 bits of multiplier at a time.

With the AMO and with reasonable modifications in the present multiplication schemes,[4] additional speeds over present methods of multiplication can be attained. However, for high speed gains, we feel that the general multiplication schemes suggested in this report should add another dimension to the organization of arithmetic units of future computer systems. An important feature of this suggested scheme is its natural ability to match with pipe line computer systems.

ACKNOWLEDGMENT

The authors wish to thank Messrs. M. S. Axelrod and G. A. Maley for their encouragement and discussions during the course of this work.

REFERENCES

1 I FLORES
The logic of computer arithmetic
Prentice-Hall Inc 1963

2 O L MacSORLEY
High speed arithmetic in binary computers
IBM Tech Report RE 00 740 Oct 1960

3 G A MALEY E J SKIKO
Modern digital computers
Prentice-Hall Inc 1964

4 A SVOBODA
Adder with distributed control
IEEE Trans on Computer pp 749-751 Vol C-19 August 1970

Procedures for increasing fault coverage for digital networks

by L. RONALD HOOVER*

Bell Laboratories
Greensboro, North Carolina

and

JAMES H. TRACEY

University of Missouri-Rolla
Rolla, Missouri

INTRODUCTION

Definitions and assumptions

The following definition for a fault detection test (*fdt*) will be used throughout this paper: an input sequence $\mathfrak{X}$ (of length one or more) for a given network $\mathfrak{M}$ is a fault detection test for fault f^i, located in $\mathfrak{M}$, if the output response to $\mathfrak{X}$ for $\mathfrak{M}$ with no faults present and the output response to $\mathfrak{X}$ for $\mathfrak{M}$ with f^i present, differ.

Throughout this paper the abbreviation *fdt* will be used when referring to a fault detection test for a single fault; whereas, *FDT* will be used when referring to the collection of *fdt*'s or sequence of *fdt*'s which attempt to cover all faults in a network.

The methods considered within this paper are based upon the validity of the single fault assumption (*sfa*).

Scope of the problem

The problem of fault detection for combinational networks is solved by several methods.[1,2] Many of the same methods, which experience great success with combinational networks, are also very successful when dealing with synchronous networks.[3,4,5] This success can be accomplished readily when the synchronous network is considered in the space domain[6] (as compared to the time domain). In the space domain, the synchronous network obeys all the restrictions placed upon a combinational network.

With the asynchronous problem, however, success is more limited. Although some methods attempt to use space domain analysis on asynchronous networks[3,5] the results are not totally acceptable. These methods fail to produce maximum fault coverage on a general asynchronous sequential network. The reason is that due to the inequality of total delays within closed paths of an asynchronous network, the space domain model fails. The time required to generate the total fault detection test sequence for an asynchronous network by one of these methods will increase rapidly with the size of the network, the number of feedback lines, and the levels over which the feedback is passed.

When considering the sequential problem, the asynchronous case is of most interest since it is more general. Faults within a synchronous network may yield a network which does not obey the restrictions placed upon the general synchronous model.

Since it is usually assumed that the general asynchronous problem defies closed solution, this research has been directed at developing supplementary procedures which can be used in conjunction with any *FDT* to increase the fault coverage yield. It is intended that these supplementary procedures, hereafter referred to as secondary techniques, would be utilized in the design stage of packaged digital components.

In the light of *LSI* technology, it is apparent that a small increase in actual gate count within a network does not significantly increase the package cost. The secondary techniques presented herein utilize this fact while trying to hold to a minimum the number of

* Previously at the University of Missouri-Rolla, Missouri.

additional external package contacts required, a factor which greatly influences cost, to facilitate greater fault coverage.

The secondary techniques will also cover faults within redundant network elements. Failure to handle redundancies is one major shortcoming of existing methods. Friedman[7] has shown that this shortcoming can lead to the concealing of otherwise detectable faults.

Two methods will first be presented which lead to modification of the package to increase the fault coverage. A technique is then presented which facilitates coverage of faults which are otherwise undetectable under the application of $\mathcal{X}$ to $\mathcal{M}$. This technique results in a modification of the original $\mathcal{X}$ sequence.

The problem of generation of the original *FDT* sequence $\mathcal{X}$ for $\mathcal{M}$ is not treated by this paper. It is assumed that an *FDT* is available and may have been generated by modified *d*-algorithm, boolean difference, or some other technique. However, since $\mathcal{M}$ may be asynchronous and observation is limited to primary outputs, in the general case $\mathcal{X}$ will not detect all of the single logical faults within $\mathcal{M}$.

Preliminary notation

Consider M to represent the set of machines which can result from a given asynchronous network $\mathcal{M}$, being subjected to any of its possible internal single logical faults. That is, if $F=(f^1,f^2\ldots n^n)$ is the set of all possible single logical faults of $\mathcal{M}$, then $M=(m^0, m^1, m^2\ldots m^n)$ is the set which corresponds to the $n+1$ configurations of the network $\mathcal{M}$ in the presence of the elements of F. That is, for each f^i contained in F there exists a unique m^i contained in M. The element m^0 will be used to represent the network $\mathcal{M}$ in the fault free configuration.

Allow Md (d for detected) to represent the set of machines such that for each m^i contained in Md, $\mathcal{Z}^i$ (where $\mathcal{Z}^k$ represents the output sequence of m^k under the application of $\mathcal{X}$). A parallel definition exists for Mu (u for undetected). Thus, the application of $\mathcal{X}$ to $\mathcal{M}$ partitions M into two disjoint subsets, Md and Mu. Since the mapping from the set $(M-m^0)$ to the set F is one-to-one and onto, there exists a similar partitioning on F. That is, Fd will represent *d*etected faults and Fu *u*ndetected faults. The sets Mu and Fu will be of concern here.

METHOD 1

The analysis of method 1 utilizes the simulator data which results from a simulation of the network in its fully faulted configuration, under the application of the input sequence $\mathcal{X}$. The theory involved with this method is presented in Appendix A.

In the effort to increase fault coverage, method 1 considers all internal signal lines as candidates for direct monitoring. Any signal line which, under direct monitoring can detect one or more of the faults contained in the set Fu is a candidate for direct monitoring. After this analysis has been performed for all internal signal lines there exists a set of signal lines, each element of which was effective in increasing the fault coverage.

This set of signal lines is then subjected to a classic cover analysis. The result of this cover analysis will be a set of signal lines S_s which has a near minimum number of elements and yields maximum increase in fault coverage. When using method 1 the designer must then make each signal line contained in S_s a primary output for testing. On some networks it was found that this method was more desirable than method 2, however even though the set S_s can sometimes be optimized further this method is generally inferior to method 2.

Two major disadvantages to method 1 are:

(1) simulation of the fully faulted network is required, and
(2) there is no upper bound placed upon the number of additional primary outputs required by this method.

METHOD 2

This method performs analysis on data provided by a fault free simulation of the network under the application of the $\mathcal{X}$ sequence.The set of undetected faults Fu is partitioned into Fu_0 and Fu_1 to correspond to undetected *sa*0 and *sa*1 type faults. The data for those signal lines which can be associated with faults of Fu_0 are subjected to a time domain analysis to determine the set of all signal lines on which individual faults of Fu_0 can be detected simultaneously. This is done for all time periods associated with a change on the input vector. The resulting sets associated with these time intervals can be logically connected to a single output pin, $\Phi(0)$.

A parallel analysis is performed for the signal lines associated with the faults of the set Fu_1. The result here is an additional output pin $\Phi(1)$.

The result from this method is two new output pins, one associated with *sa*0 faults, the other with *sa*1 faults, which yield maximum fault coverage under the input sequence $\mathcal{X}$. The theoretical development of the algorithm which generates the Φ functions is presented in Appendix B.

In addition to the two primary outputs required by this method, two additional inputs, I_0 and I_1, are needed to facilitate the detection of faults within the $\Phi(0)$ and $\Phi(1)$ networks.

The major advantages to method 2 are: (1) only fault free simulation is required, (2) regardless of the size of the network, a maximum of four additional external contacts is required.

Several techniques have been discussed for decreasing the maximum below 4 additional contacts discussed above.[10]

SEQUENCE MODIFICATION

Under application of $\mathcal{X}$, the additional external contacts from methods 1 or 2, when considered as new primary outputs, partition Mu into two disjoint subsets, Mu_d (*d*etected) and Mu_u (*u*ndetected). A similar partition exists on Fu; that is, for each m^i contained in Mu_d, then f^i contained in Fu_d and for each m^j contained in Mu_u, then f^j contained in Fu_u. If the external contacts, which have been added to facilitate this partition are considered to be the r components of an output vector P, then for the application of $\mathcal{X}$ on $\mathcal{M}$ the results are:

(1) for each m^i contained in Mu_d, $\rho^i \neq \rho^0$ (where ρ^k is the output sequence of P vectors from m^k under application of $\mathcal{X}$).
(2) for each m^j contained in Mu_u, $\rho^j = \rho^0$.

Application of the *FDT* sequence $\mathcal{X}$ to $\mathcal{M}$ has been successful in detecting all single faults except those which result in the set Mu_u. Since these faults could not be detected by direct monitoring of the signal line, it is apparent that under the application of $\mathcal{X}$ to $\mathcal{M}$, the signal line associated with fault f^i, for each f^i contained in Fu_u, did not assume the proper value to allow for detection of f^i. As an example, to facilitate detection of the fault, line a ($sa1$), the *FDT* sequence must force line a in m^0 to assume the value 0 at least once. The problem is to develop a heuristic which will allow modification of $\mathcal{X}$ so as to enable detection of the faults f^i contained in Fu_u. The heuristic technique presented here borrows on the theory which has developed around the use of the classic d-algorithm.[1] The similarity will be seen between this method and the consistency test or backward drive segment of the d-algorithm.

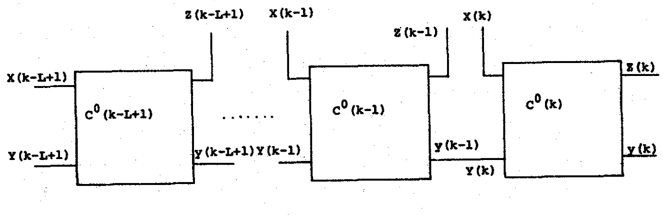
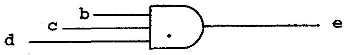


Figure 1—General space domain model



singular cover

b	c	d	e
1	1	1	1
0	x	x	0
x	0	x	0
x	x	0	0

Figure 2—Singular cover for an AND gate

Following Breuer[6] it is suggested that the time domain analysis of the system $\mathcal{M}$ be mapped into its corresponding special equivalent. This mapping can be accomplished if, for each new input vector, a new copy of $\mathcal{M}$ is allowed. Since it is the goal to force a given value on a particular line in m^0, the multiple copies of m^0 will be labeled $C^0(k)$, $C^0(k-1)$, - - - - $C^0(k-L+1)$. The length L of the new sequence $\mathcal{X}$ m^i generated in this manner can be dynamically determined within reasonable restraints. The space domain analysis can be understood by observing Figure 1.

The copies of the machine are interconnected in such a way that in addition to the original input vector, $C^0(k-d)$ has as inputs on its $Y_{(k-d)}$ lines the state variable vector $y_{(k-d-1)}$ from copy $C^0(k-d-1)$.

Assume that it is necessary to generate an input sequence $\mathcal{X}$ m^i of length L to aid in detecting f^i contained in Fu_u, a $sa1$ fault on line a. First, assign line a in $C^0(k)$ the value 0 and attempt to drive this signal from $C^0(k)$ back through all copies to $C^0(k-L+1)$.

The method for accomplishing the backward drive will now be discussed. For all gates along the signal paths which control line a of $C^0(k)$, the singular covers[8] must be formed. An example of the singular cover for a 3 input AND gate is given in Figure 2.

The singular cover for $C^0(k)$ is formed between inputs and signal line a. The required value on line a is then driven backward to the inputs of $C^0(k)$ by performing intersections on the singular covers of the gates along the path. All parallel paths must be intersected simultaneously. However, intersections need not be made with singular cover vectors for gates whose outputs are unrestricted. The rules for intersection are:

$$1\Lambda 0=\emptyset=0\Lambda 1$$

$$x\Lambda 0=0=0\Lambda x$$

$$x\Lambda 1=1=1\Lambda x$$

If at any time during the backward drive a $\emptyset$ results, then an inconsistency exists and a retrace is required beginning with a new vector from the appropriate singular cover.

If $\mathfrak{M}$ is asynchronous, care must be taken when picking vectors from the singular cover for intersection. It must be assured that $D[X(k-r)-X(k-r+1)]\leq 1$ (where D is the Hamming inter-vector distance). As an example, if $X(k-2)=[0xx1]$ and $X(k-1)=[01x1]$, $D=1$. This, however, may force the reevaluation of $D[X(k-1)-X(k)]$.

When the backward drive to the inputs of $C^0(k)$ is completed, the values required on the input vectors $X(k)$ and $Y(k)$, which is being input from the $C^0(k-1)$ copy, is $Y(k)=[xxx\ldots.x]$ (unrestricted), then the result is a sequence $\mathfrak{X}m^i$ of length $L=1$. However, if $Y(k)\neq[xxxx\ldots x]$, the backdrive must continue through $C^0(k+1)$. This procedure continues until at some level $(k-L+1)$, $Y(k-L+1)=[xxx\ldots x]$. This strategy is required so that the sequence which is generated is not state dependent. Therefore, the sequence $\mathfrak{X}m^i$ is forced to produce the desired result on line a regardless of the state of $\mathfrak{M}$ when $\mathfrak{X}m^i$ is applied. If, due to network configuration, information concerning machine state is known, this requirement can be appropriately relaxed. If at the $(k-r)$ level the condition $Y(k-r)=[xx\ldots..x]$ is not satisfied, the procedure must continue to the $(k-r-1)$ level. However, this process must not be allowed to continue indefinitely. One criterion for stopping the process short of success would be to determine some cost effective constant R and require that $L\leq R+1$.

If this technique yields a sequence $\mathfrak{X}m^i$ and if $\mathfrak{M}$ is synchronous or combinational, $\mathfrak{X}m^i$ is certain to assign the proper value to line a; that is if $\mathfrak{X}m^i=X(k-L+1)$, $X(k-L+2),\ldots X(k-1)$, $X(k)$ is applied to m^0 beginning at time $t=t_0$, line a will assume the desired value at $t=t_0+L$ (with L assigned time units). If $\mathfrak{M}$ is asynchronous, the space domain model fails; thus, the technique is heuristic, and $\mathfrak{X}m^i$ must be simulated to check on its validity. In either case, if $\mathfrak{X}m^i$ is valid, the new FDT, which covers the set of faults, (f^i+F-Fu_u), is $\mathfrak{X}\mathfrak{X}m^i$. That is, $\mathfrak{X}m^i$ concatenated to the end of $\mathfrak{X}$. If there are other faults, f^j contained in Fu_u, which are not covered by $\mathfrak{X}\mathfrak{X}m^i$ then this procedure would be repeated for f^j. There is no guarantee that the $\mathfrak{X}m^i$ found in this manner is optimal. The length of $\mathfrak{X}m^i$ is dependent upon the choice of vectors from the singular covers.

After all sequence modifications of the form $\mathfrak{X}m^j$ have been produced, the total modifications are then simulated with $\mathfrak{X}$, to determine their success. If the $\mathfrak{X}m^j$'s are successful these results must be combined with either method 1 or method 2.

CONCLUSION

Summary

Two techniques have been presented which yield modification to the general digital network to facilitate maximum fault coverage under a given input sequence. Method 2 accomplishes the network modifications with a minimum impact upon the surrounding environment with which the network must interface.

The technique for providing input sequence modifications will have little value if the original input sequence was designed by an accepted fault detection test generation algorithm. However, if the original $\mathfrak{X}$ sequence was developed by a less effective technique and if $\mathfrak{M}$ lends itself to space domain analysis, this technique is very useful. It seems evident that if the designer purposely exercises the trade offs made available by these techniques, an acceptable level of fault coverage can be realized on any general digital network. Although these techniques are useful on all types of networks, it seems apparent that they are of extreme importance in the asynchronous sequential area since it is in this area that previous techniques fail.

Results

The TEGAS[9] digital logic simulator was utilized in collecting data to evaluate the secondary techniques. This system is implemented on an IBM 360/50 system in Fortran IV and can simulate 32 different network fault configurations with each pass through the network.

The simulator presents the network data in a form which is readily usable by the secondary techniques. The signal line values can be readily interrogated at any time to determine fault coverage. Although some of the actual data analysis for the secondary techniques was done manually, this process is being program implemented and interfaced with the TEGAS simulator.

The computer run time required by the simulator is dependent, not only upon the element count for the network, but also upon network structure. Typically, asynchronous networks with 15-30 elements will require 1-5 minutes of computer time for simulation with an input sequence of length 10. It is expected that when the secondary techniques have been program implemented and interfaced, this time will increase by something less than 35 percent.

APPENDIX A

This appendix will present the theoretical foundation underlying method 1.

Consider the set of all signal lines contained in $\mathfrak{M}$ to be $S=(s_1, s_2, \ldots\ldots s_m)$. S contains all primary inputs, primary outputs, feedback lines, and all internal connection lines. For each $s_i \in S$, two logical faults can be associated; that is, $s_i(Sa0)$ and $S_i(Sal)$. The total number of faults can be collapsed across each network element; but since this in no way influences the theory of solution, it will be ignored until it can be utilized to expedite data analysis.

For each $s_i \in S$ there exists $f^i \in F$ and $f^j \in F$ and $m^i \in M$ and $m^j \in M$. Observation of the output sequence $\mathcal{Z}=z_1z_2z_3 \ldots\ldots\ldots z_w$, for the application of $\mathfrak{X}=X_1X_2X_3 \ldots\ldots\ldots X_w$ to $\mathfrak{M}$ performs a partitioning of M and F. This partitioning can be applied to the set S. Consider the set Su (*u*ndetected) to represent the set of signal lines such that $\forall s_i \in Su$ there exists at least one $f^j \in Fu$ corresponding to a logical fault on s_i. Sd will be the subset such that $\forall s_j \in Sd$ there exists exactly two faults, f^k and f^l, $\in Fd$ which are associated with faults on signal line s_j.

The value on signal line s_i after the application of X_k, in the $\mathfrak{X}$ sequence, to machine m^j, will be represented by $v(i, j, k)$. For the application of each input vector X_k, in the $\mathfrak{X}$ sequence, first a comparison of $v(i, 0, k)$ with $v(i, j, k)$ is made for all j to determine which elements of M can be detected by s_i under application of X_k. This must be done $\forall s_i \in S$. This entire process must then be performed for $k=1$ to w. The result from this operation will be a set of fault coverage lists of the form s_i, X_k, m^p, $m^l, \ldots m^r$, where this list represents the fact that by observing line s_i, while X_k, in the $\mathfrak{X}$ sequence, is applied to $\mathfrak{M}$, faulty machines $m^p, m^l, \ldots\ldots m^r$ can be detected. It is upon these fault coverage lists that the cover analysis must be performed to determine which signal lines must be monitored.

The rules for performing the cover analysis will now be considered. All signal lines which are primary outputs are, by definition, going to be monitored. Consider the set of all primary output lines to be S_z. For each $s_i \in S_z$, s_i is a primary output of $\mathfrak{M}$. Thus, the removal of all faults which are associated with the fault coverage lists of the elements of S_z before the analysis starts is necessary. $\forall s_i \in S_z$, there is associated a set of fault coverage lists of the form s_i, X_k, m^p, $m^l \ldots m^r$. By combining all machines which are listed in the fault coverage lists for signal lines s_i the set Mz_i is formed, where $\forall m^j \in Mz_i$, m^j can be detected by monitoring s_i. Similar sets Mz_k are formed $\forall k$ such that $s_k \in S_z$. It can be seen that the set $Md=U(Mz_i)$ for all i such that $s_i \in S_z$ (where U is the set union operation). In a similar fashion, sets Ms_i for all i, such that, $s_i \in (S-S_z)$ are formed. From each such set Ms_i, the elements which are common to Ms_i and Md are then removed. That is, $Ms_i^*=Ms_i-(Ms_i \Lambda Md)$ is formed (where Λ is a set intersection operation). There now exists a set of the sets of form Ms_i^*, where $\forall\ m^j \in Ms_i^*$, $m^j \in Mu$ and m^j can be detected by monitoring s_i. To decide which signal lines of the set $(S-S_z)$ must be monitored, first a search for critical signal lines is performed. That is, $\forall m^i \in Mu$, for which m^i is contained in one and only one Ms_j^*, monitoring of s_j is required. All machines which are covered by any such line s_j must now be removed from the Ms_k^* for all remaining lines in $(S-S_z)$. The cover analysis then proceeds using the following two rules:

(1) The signal line with the highest value is the next line entered into the set S_s. The value for any line is equal to the number of previously undetected faults which are covered by monitoring this line.

(2) If several lines have equal value, the choice will be arbitrary with the only priority being assigned to state variable lines.

The results of this analysis will be two sets of signal lines S_z and S_s, where $\forall s_i \in S_z$, s_i is a primary output and $\forall s_k \in S_s$, s_k is not a primary output.

The members of S_s are the signal lines which will require additional primary outputs from the package to facilitate monitoring.

If $\mathfrak{M}$ represents a general network, then $\forall s_k \in S_s$, it is necessary to add an additional primary output.

APPENDIX B

This appendix presents the theoretical foundation underlying Method 2.

The set M is partitioned into Md and Mu by the application of $\mathfrak{X}$ to $\mathfrak{M}$. The elements of each Mu and Fu are then further partitioned into two disjoint subsets—Fu_0, Mu_0 and Fu_1 and Mu_1—where $\forall m^i \in Mu_0$, the associated $f^i \in Fu_0$ is a *sa*0 type logical fault, and $\forall m^j \in Mu_1$, the associated $f^j \in Fu_1$ is a *sa*1 type logical fault. For each fault $f^i \in Fu_0$, there is an associated signal line s_k. S_0 will be the set of signal lines associated with the faults of Fu_0 and similarly S_1 and Fu_1. Since, in general, we may have both logical faults f^i and f^j associated with a given line as elements of Fu, generally, $S_1 \Lambda S_0 \neq \emptyset$. The signal lines s_i, such that $s_i \in (S_1 U S_0)$, are the lines which must be monitored. If under the input vector X_k from $\mathfrak{X}$, the signal line s_i (where $s_i \in S_0) = 1$ in m^0, then s_i can be monitored to detect f^i (where $f^i \in Fu_0$ is one of the faults associated with s_i) during S_k. Since there may be many such s_i's for a given X_k, there will be associated with each input vector two sets of signal lines, $SX_k(0)$ and $SX_k(1)$ where $\forall s_i \in SX_k(0)$ the fault f^i (where $f^i \in Fu_0$ is a fault associated with s_i) can be detected by monitoring s_i during X_k. Likewise, $\forall s_j \in SX_k(1)$, the fault f^j (where $f^j \in Fu_1$ is one of the faults associated with line s_j) can be detected by monitoring line s_j during X_k. After the entire sequence has been applied to $\mathfrak{M}$ and all of the sets of the type $SX_k(\alpha)$ have been formed, a set $\mathcal{S}(0) = (SX_k(0),\ SX_{k+r}(0) \ldots\ldots)$ is formed. $\mathcal{S}(0)$ is formed by including sufficient elements $SX_k(0)$ so that $\forall s_i \in S_0$, for which there exists at least one $SX_k(0) \in \mathcal{S}(0)$. Thus $f^i \in Fu_0$ can be detected by monitoring s_i during X_k. Similarly $\mathcal{S}(1) = (SX_s(1),\ SX_{s+r}(1) \ldots\ldots)$.

The following notation is now defined. If we have a set $R = (r_1, r_2, r_3, \ldots\ldots r_k)$, then $\Pi(R) = \Pi(r_1,\ r_2,\ \ldots r_k) = (r_1 \cdot r_2 \cdot r_3 \cdot \ldots\ldots r_k)$, where $(\cdot)$ represents the logical AND operation. Similarly, $\Sigma(R) = (r_1, r_2 ---- r_k) = (r_1 + r_2 + r_3 + --- r_k)$ where $(+)$ is the logical OR operation.

Utilizing the above notation, the functions

$$\Phi(0) = \sum_{\mathcal{S}(0)} [\Pi(SX_i(0), I_0]$$

$$\Phi(1) = \prod_{\mathcal{S}(1)} [\Sigma(SX_j(1), I_1] \quad \text{are formed.}$$

The I signals are conditioning signals which will be defined later. The Φ's express the logic function which must be realized on the additional network outputs so as to cover the faults of Fu which are detectable by this method.

In realizing $\Phi(0)$, it can be seen that each element of $\mathcal{S}(0)$ will define the input list to an AND gate. That is, $\forall SX_k(0) \in \mathcal{S}(0)$ there will be defined an AND gate $AX_k(0)$. Each such $AX_k(0)$ will have as inputs all elements of the set $SX_k(0)$ plus an additional conditioning signal I_0. The outputs of all such $AX_k(0)$ gates will completely define the input set for an OR gate $\Phi(0)$. The output of $\Phi(0)$ will represent one of the additional required primary outputs.

Note: This discussion has been based, for simplicity, upon two level AND-OR logic. Certainly, the type logic elements actually utilized and the method of interconnection is unrestricted so long as the function realized is unaltered.

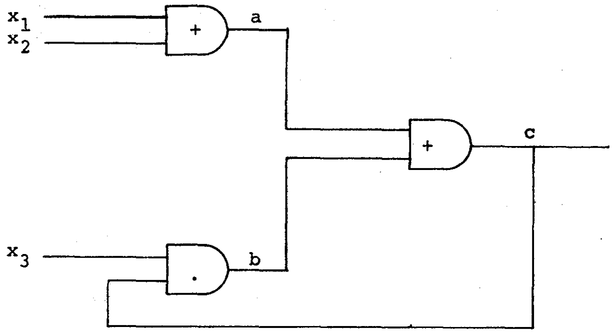


Figure 3—Example network

A similar two level OR-AND structure can be described for the $\Phi(1)$ function. Due to the parallelism between these two functions, the verbal description of $\Phi(1)$ is omitted.

The I_0 and I_1 signal lines are used to facilitate fault detection of the added hardware. $I_0 = 1$ during the application of every X_k to $\mathfrak{M}$, for which $SX_k(0) \in \mathcal{S}(0)$. $I_1 = 0$ during the application of every X_k to $\mathfrak{M}$, for which $SX_k(1) \in \mathcal{S}(1)$. It must be mentioned that if the network is such that every X_i of $\mathfrak{X}$ has associated with it an $SX_k(\alpha) \in \mathcal{S}(\alpha)$ (for $\alpha = 0$ or $\alpha = 1$), then an additional input vector must be added to $\mathfrak{X}$ to facilitate the detection of the gates in the $\Phi(\alpha)$ network. That is, if line I_α must be used to condition the gates of network $\Phi(\alpha)$ during the entire $\mathfrak{X}$ sequence, then an additional input vector must be added to $\mathfrak{X}$ so that I_α can be used to detect faults in the $\Phi(\alpha)$ network.

From the above discussion it can be seen that if the network is fault free, then $\Phi(0) = 1 \forall X_i$ for which there exists an $SX_i(0) \in \mathcal{S}(0)$. However, if we have the fault $f^j \in Fu_0$ on the signal line $s_i \in S_0$, then $\Phi(0) = 0$ for all X_k, such that $s_i \in SX_k(0)$.

TABLE I—Faulty Machine List

m^i	Specific Fault
m^1	x_1(sa1)
m^2	x_1(sa0)
m^3	x_2(sa1)
m^4	x_2(sa0)
m^5	x_3(sa1)
m^6	x_3(sa0)
m^7	a(sa0)
m^8	a(sa1)
m^9	b(sa0)
m^{10}	b(sa1)
m^{11}	c(sa0)
m^{12}	c(sa1)

A *sa* fault on the output of gate AX_k (0) of the $\Phi(0)$ network will result in $\Phi(0)=0$ during X_k. Also, $\Phi(0)\,sa0$ will be detected by $\Phi(0)=0$ during an X_k for which $SX_k(0)\in \mathcal{S}(0)$. If there exists an X_r such that $SX_r(0)\notin \mathcal{S}(0)$, then setting $I_0=0$ during X_r yields $\Phi(0)=0$ for m^0; but $\Phi(0)$ will equal 1 if any gate in the $\Phi(0)$ network is *sa*1.

A similar argument can be given for the output values and the faults within the $\Phi(1)$ network.

APPENDIX C

This section contains three example problems: Example 1 illustrates method 1, example 2—method 2, and example 3 illustrates the sequence modification technique.

Example 1

Method 2 will be illustrated using the network of Figure 3.

Table I associates with each possible single logic fault a machine number m^i.

For the input sequence $\mathcal{X}=X_1X_2X_3X_4=(111)(101)(001)(011)$, Table II shows the values of all signal lines of the network shown in Figure 1. The table includes data for the fault free and all single fault machines. Note: assume line $c=1$ at start.

TABLE II—Simulator Output Table

	Signal Lines	i = machine number (m^i) 0	1	2	3	4	5	6	7	8	9	10	11	12	Fault Coverage Lists
	x_1	1	1	0	1	1	1	1	1	1	1	1	1	1	x_1, X_1, m^2
	x_2	1	1	1	1	0	1	1	1	1	1	1	1	1	x_2, X_1, m^4
X_1	x_3	1	1	1	1	1	1	0	1	1	1	1	1	1	x_3, X_1, m^6
(111)	a	1	1	1	1	1	1	1	0	1	1	1	1	1	a, X_1, m^7
	b	1	1	1	1	1	1	0	1	1	0	1	0	1	b, X_1, m^6, m^9, m^{11}
	c	1	1	1	1	1	1	1	1	1	1	1	0	1	c, X_1, m^{11}
	x_1	1	1	0	1	1	1	1	1	1	1	1	1	1	x_1, X_2, m^2
	x_2	0	0	0	1	0	0	0	0	0	0	0	0	0	x_2, X_2, m^3
X_2	x_3	1	1	1	1	1	1	0	1	1	1	1	1	1	x_3, X_2, m^6
(101)	a	1	1	0	1	1	1	1	0	1	1	1	1	1	a, X_2, m^2, m^7
	b	1	1	1	1	1	1	0	1	1	0	1	0	1	b, X_2, m^6, m^9, m^{11}
	c	1	1	1	1	1	1	1	1	1	1	1	0	1	c, X_2, m^{11}
	x_1	0	1	0	0	0	0	0	0	0	0	0	0	0	x_1, X_3, m^1
	x_2	0	0	0	1	0	0	0	0	0	0	0	0	0	x_2, X_3, m^3
X_3	x_3	1	1	1	1	1	1	0	1	1	1	1	1	1	x_3, X_3, m^6
(001)	a	0	1	0	1	0	0	0	0	1	0	0	0	0	a, X_3, m^1, m^3, m^8
	b	1	1	1	1	1	1	0	1	1	0	1	0	1	b, X_3, m^6, m^9, m^{11}
	c	1	1	1	1	1	1	0	1	1	0	1	0	1	c, X_3, m^6, m^9, m^{11}
	x_1	0	1	0	0	0	0	0	0	0	0	0	0	0	x_1, X_4, m^1
	x_2	1	1	1	1	0	1	1	1	1	1	1	1	1	x_2, X_4, m_4
X_4	x_3	1	1	1	1	1	1	0	1	1	1	1	1	1	x_3, X_4, m^6
(011)	a	1	1	1	1	0	1	1	0	1	1	1	1	1	a, X_4, m^4, m^7
	b	1	1	1	1	1	1	0	1	1	0	1	0	1	b, X_4, m^6, m^9, m^{11}
	c	1	1	1	1	1	1	1	1	1	1	1	0	1	c, X_4, m^{11}

From Table II, it can be seen that since

$$Md=[m^6, m^9, m^{11}] \quad \text{then}$$

$$Mx_1^*=[m^1, m^2]$$

$$Mx_2^*=[m^3, m^4]$$

$$Mx_3^*=\emptyset$$

$$Ma^*=[m^1, m^2, m^3, m^4, m^7, m^8]$$

$$Mb^*=\emptyset$$

The cover analysis is shown in Table III.

From Table I it can be seen that by monitoring signal line *a*, all faults coverable by this method are detected. By monitoring line *a* along with the primary output *c*,

TABLE III—Cover Analysis
Elements of Mu

		m^1	m^2	m^3	m^4	m^5	m^7	m^8	m^{10}	m^{12}
Signal	x_1	x	x							
Lines	x_2			x	x					
	a	x	x	x	x		x	x		

all machines, except m^5, m^{10}, and m^{12}, can be detected. Faults f^5, f^{10}, and f^{12} are undetectable under this input sequence.

Example 2

Referring to the network of Figure 3, the following sets are enumerated to further clarify the theoretical discussion included in Appendix B concerning method 2.

$$Md=[m^6, m^9, m^{11}]$$

$$Mu=[m^1, m^2, m^3, m^4, m^5, m^7, m^8, m^{10}, m^{12}]$$

$$Mu_0=[m^2, m^4, m^7]$$

$$Mu_1=[m^1, m^3, m^5, m^8, m^{10}, m^{12}]$$

$$Fu_0=[x_1(sa0), x_2(sa0), a(sa0)]$$

$$Fu_1=[x_1(sa1), x_2(sa1), x_3(sa1), a(sa1), b(sa1), c(sa1)]$$

$$S_0=[x_1, x_2, a]$$

$$S_1=[x_1, x_2, x_3, a, b, c]$$

Table IV contains the fault free simulation data. From

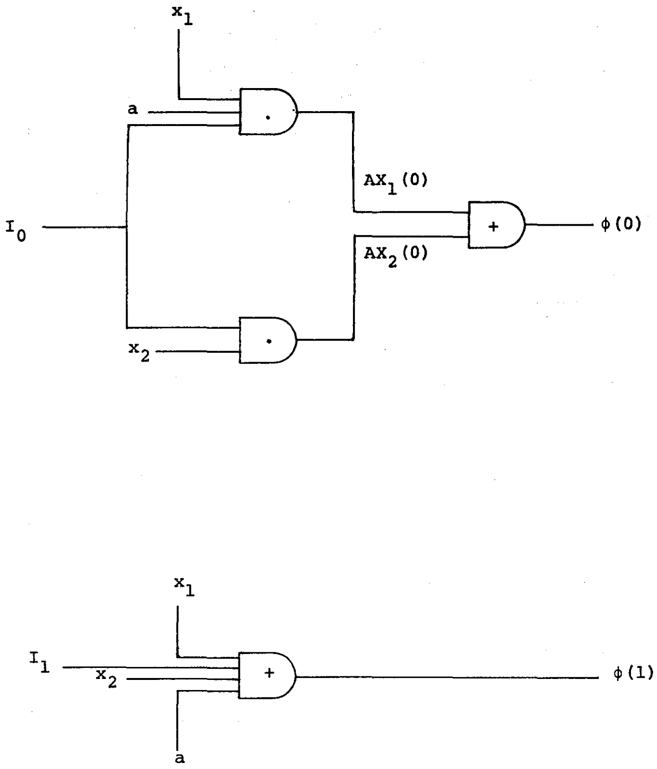


Figure 4—Networks leading to additional outputs

TABLE IV—Fault Coverage Table

	Signal			S_0				S_1				
	Lines	m^0	x_1	x_2	a	x_1	x_2	x_3	a	b	c	
	x_1	1	x									
	x_2	1		x								
x_1 = (111)	x_3	1										SX_1 (0) = $[x_1,x_2,a]$
	a	1			x							SX_1 (1) = ∅
	b	1										
	c	1										
	x_1	1	x									
	x_2	0					x					
x_2 = (101)	x_3	1										$XS_2(0)$ = $[x_1,a]$
	a	1		x								$XS_2(1)$ = $[x_2]$
	b	1										
	c	1										
	x_1	0				x						
	x_2	0					x					
x_3 = (001)	x_3	1										$SX_3(0)$ = ∅
	a	0									x	$SX_3(1)$ = (x_1, x_2, a)
	b	1										
	c	1										
	x_1	0				x						
	x_2	1	x									
x_4 = (011)	x_3	1										$SX_4(0)$ = (x_2)
	a	1										$SX_4(1)$ = (x_1)
	b	1										
	c	1										

TABLE V—Singular Cover for Gate $b(k)$

$x_3(k)$	$c(k-1)$	$b(k)$	label
1	1	1	A
0	x	0	B
x	0	0	C

this table it can be seen that

$$\mathcal{S}(0)=(SX_1(0)) \text{ or}$$

$$\mathcal{S}(0)=(SX_2(0)),\ (SX_4(0))$$

$$\mathcal{S}(1)=(SX_3(1))$$

The networks which lead to outputs $\Phi(0)$ and $\Phi(1)$ are shown in Figure 4. $\mathcal{S}(0)=[SX_2(0),\ SX_4(0)]$ is used to give an example of a two level result.

Example 3

To illustrate the sequence modification technique, an example follows based upon the network of Figure 3.

Assume that signal $b(sa0)$ is a fault which has not been detected. It is necessary to force a logical 1 on b. This procedure begins by turning to the space domain analysis and forming the singular cover of the network from line b to the inputs of copy $C^0(k)$. The space domain model is shown in Figure 5.

Table V shows the singular cover vectors for $b(k)$ in $C^0(k)$.

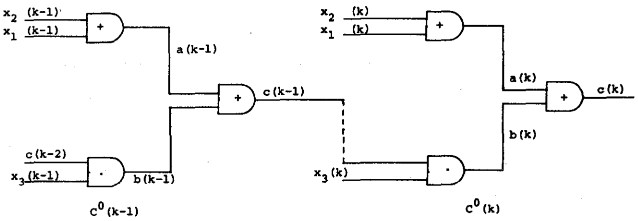


Figure 5—Space domain model of Figure 3

Since the feedback line $c(k-1)\neq x$ when $b(k)=1$, the process must proceed to the $(k-1)$ level. Therefore, $C^0(k-1)$ is added to Figure 5 and the singular covers listed in Table VI are formed.

The singular cover vector A from $b(k)$, labeled $A_{b(k)}$, can be intersected with either A or B of the singular cover of $c(k-1)$. Since b is the gate which is influenced directly by the feedback line, the intersection between $A_{b(k)}$ and $B_{c(k-1)}$ is performed. This intersection will place less restrictions on the feedback line which is input to gate $b(k)$. The results of the intersections are shown in Table VII. A^* need not be intersected with any of the singular covers of $b(k-1)$ since $b(k-1)=[x]$. A^* is now intersected with either $A_{a(k-1)}$ or $B_{a(k-1)}$. The result is shown for $A_{a(k-1)}$. This final vector has $Y(k-1)=c(k-2)=[x]$. Therefore, the procedure stops with $L=2$. The $\mathcal{X}m^i$ sequence is $X_1X_2=(x1x)(xx1)$. It can be verified by hand simulation that this sequence does indeed force line b to have a value 1.

TABLE VI—Singular Covers for the Gates of Figure 5

C⁰(k − 1)			C⁰(k)					
x₁(k − 1)	x₂(k − 1)	c(k − 2)	x₃(k − 1)	a(k − 1)	b(k − 1)	c(k − 1)	label	gate name
				x	1	1	A	
				1	x	1	B	c(k − 1)
				0	0	0	C	
		1	1		1		A	
		0	x		0		B	b(k−1)
		x	0		0		C	
x	1			1			A	
1	x			1			B	a(k−1)
0	0			0			C	

TABLE VII—Intersection Table

x₁(k−1)	x₂(k−1)	c(k−1)	x₃(k−1)	a(k−1)	b(k−1)	c(k−1)	x₃(k)	b(k)	label	description
				1	x	1	1	1	A*	$A_{b(k)}\,\Lambda B_{c(k-1)}$
x	1			1	x	1	1	1	B*	$A^*\Lambda A_{a(k-1)}$

REFERENCES

1 J P ROTH
Diagnosis of automata failures: A calculus and a method
IBM J Res Develop Vol 10 pp 278-291 July 1966

2 M Y HSIAO D K CHIA
Fundamentals of boolean difference for test pattern generation
Proc 4th Annual Princeton Conf Inform Sci March 1970

3 G R PUTZOLA J P ROTH
A heuristic algorithm for the testing of asynchronous circuits
IEEE Trans on Elec Comp Vol C-20 pp 639-647 June 1971

4 M Y HSIAO D K CHIA
Boolean difference for fault detection in asynchronous sequential machines
IEEE Trans on Elec Comp Vol C-20 pp 1356-1361 Nov 1971

5 W G BOURICIUS et al
Algorithm for detection of faults in logic circuits
IEEE Trans on Elec Comp Vol C-20 pp 1258-1264 Nov 1971

6 M A BREUER
A random and an algorithmic technique for fault detection test generation for sequential circuits
IEEE Trans on Elec Comp Vol C-20 pp 1364-1370 Nov 1971

7 A D FRIEDMAN
Fault detection in redundant circuits
IEEE Trans on Elec Comp Vol EC-16 pp 99-100 1967

8 H V CHANG E MANNING G METZE
Fault diagnosis of digital systems
New York John Wiley and Sons 1970 pp 29–47

9 D M ROUSE
A simulation and diagnosis system incorporating various time delay models and functional elements
PhD Dissertation University of Missouri-Rolla Rolla Missouri 1970

10 L R HOOVER
Secondary techniques for increasing fault coverage of fault detection test sequences for asynchronous sequential networks
PhD Dissertation pp 45-64 University of Missouri-Rolla Rolla Missouri 1972

System identification and simulation—A pattern recognition approach*

by W. J. KARPLUS

University of California
Los Angeles, California

INTRODUCTION

Recent years have seen continuing and increasingly-intensive attempts to extend the art of simulation to areas which heretofore were considered too complex and too difficult to lend themselves to conventional modelling and simulation techniques. These include such environment-oriented fields as air-pollution, water conservation, thermal pollution, etc., as well as systems belonging to the biological, the medical, the economic, and sociological areas. For example, in 1970 the Office of Water Resources Research catalogued over 600 on-going research projects concerned with the modelling of water resource systems. The extension of simulation techniques developed in application areas such as control system design, electro-mechanical systems, etc., to these new areas has often been disappointing, if not completely unsuccessful. This is due to the difficulty in constructing a sufficiently-valid mathematical model—a model which can be used for prediction with a reasonable amount of confidence. It is well-known, of course, that even under the best conditions, inverse problems such as system identification problems, do not have unique solutions. That is, inevitably an infinite number of possible models will satisfy a specified set of excitation/response relationships. Where the identification process is further handicapped by uncertainties as to system structure and inadequate experimental data, the pertinent question is often not: "How good is the model?" but rather: "Is there any point to modelling at all?"

It is the purpose of the present paper to suggest an approach to the derivation and utilization of mathematical models. This approach may be considered to be a generalization and formalization of what has broadly been called "gaining insight into the operation of a system," but is particularly intended to assist in the formulation of relatively-valid mathematical models for subsequent simulation and in the specification of what additional data (in the form of observations and measurements of the system to be modelled) must be provided to permit meaningful modelling. To put the discussion into proper perspective, the conventional approach to modelling and simulation is first summarized, followed by a discussion of the weaknesses of this method when applied to highly-complex systems. The pattern recognition method is then outlined.

CLASSES OF MODELLING PROBLEMS

A variety of techniques is available for the analysis of systems. Where analytical solution and direct experimentation upon a system are impractical, recourse is often made to simulation. Simulation is a numerical technique which takes the following steps:

1. A system existing in the real world (the prototype system) is represented by a *model*. This model usually characterizes some interesting facets of the prototype system behavior by a set of equations.
2. The model is *implemented* or programmed on a computer in such a manner that system inputs, system parameters and perhaps system structure can be conveniently varied and the effects of these variations studied.
3. The computer is employed to perform a variety of *experiments* so as to provide the information that constitutes the basic objectives of the simulation. These experiments usually involve the prediction of the behavior of the prototype system under various conditions.

The necessary first step in any simulation is, therefore, the formulation and validation of the model. The

* The work described in this paper is supported by the National Science Foundation under Grant GK31463X.

discussion of system modelling and the development of a comprehensive modelling philosophy is handicapped by the fact that the types of models used, the raw information available to assist in the development of the model, and the objectives of the eventual simulation depend strongly upon the specific application area and upon subfields within specific areas. The following distinctions are particularly important.

In some applications, the modelling is of the so-called "black-box" variety, in which there is virtually no *a priori* knowledge as to the nature and structure of the equations characterizing dynamic system behavior. In other applications, the problem is one of modelling a "gray-box" of various shades of gray; here one starts from a knowledge of the general nature of the mathematical model and is concerned primarily with the determination of certain system parameters, initial conditions, and structural details. In some identification problems, the excitation/response data used to identify the gray-box are accurately known or specified and are, therefore, dependable starting points for the modelling process; in other situations, the data to be used to identify systems are obtained by measurements on the system to be modelled and may be seriously corrupted by noise, sampling errors, and a variety of shortcomings in the information gathering effort. In that case, the term "estimation" is often used instead of the term "identification." Finally, a distinction must be made between systems in which the excitation/response observations are made "actively" and those which utilize "passively" obtained data. In active system identifications, experimental data are collected by subjecting the system to be identified to a series of systematic tests involving the application of specified excitations and observing in each case the resulting response. In passive identifications, on the other hand, the analyst is limited to using data which are generated in response to excitations over which the analyst exercises no control and is, therefore, impeded from constructing key experiments to aid in the identification effort. Clearly the above considerations, that is how black the box, the extent to which excitation/response data are free from noise, and the extent to which active data gathering is possible, greatly affect the modelling process and the reliability and validity of the resulting model.

The present discussion is focused on the modelling of a class of systems which are of interest in a variety of environmental studies and similar large-scale system problems. Consider, for example, the problem of modelling an underground water reservoir or aquifer.[1,2] The water inputs to the underground porous medium, including rainfall and underground streams, are approximately known, and the underground water level has been measured and recorded at a number of wells which have been drilled into the aquifer. It is desired to obtain a model and perform a simulation to assist in the determination of an optimum control policy to specify how much water can be withdrawn from the various wells so as to maintain the water level in the aquifer at a desired level. It is known that the fluid flow in a porous medium obeys the nonlinear parabolic partial differential equation,

$$\frac{\partial}{\partial x}\left(T(x, y, h)\,\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial y}\left(T(x, y, h)\,\frac{\partial h}{\partial y}\right)$$

$$=S(x, y, h)\,\frac{\partial h}{\partial t} - Q(x, y, t) \quad (1)$$

where

$h=h(x, y, t)$ = hydraulic potential at any point (x, y) on the water table at time t and approximately represents the elevation of the water table above a reference plane.

$Q=Q(x, y, t)$ = accretion to the water table due to rainfall, lateral flow, wells, etc.

$T=T(x, y, h)$ = transmissivity, which is a measure of the fluid conductivity of the aquifer.

$S=S(x, y, h)$ = storage coefficient, which is a measure of the fluid capacity of the aquifer.

The nonlinear parameters $T(x, y, h)$ and $S(x, y, h)$ are governed by local soil characteristics and are largely unknown. Likewise, the geometry of the field, the location of the boundaries of the aquifer in the x-y plane, is only very approximately known from geologic explorations. The only information available to permit an inference of these unknown functions are the well histories (the height of the water measured every few months) taken at a number of wells over a period of years. These represent the raw data. It is the objective of the system identification process to derive a mathematical model including particularly a specification of T and S and of the boundary configuration.

In terms of the distinctions briefly discussed above, the modelling effort for problems of this type involves the solution of a "gray-box" problem since it is usually more-or-less accepted that the dynamic processes under study are characterized by nonlinear partial differential equations, including two or three space-variables and time as independent variables, in which the field parameters must be determined by the identification procedure. The response data are noisy, subject to

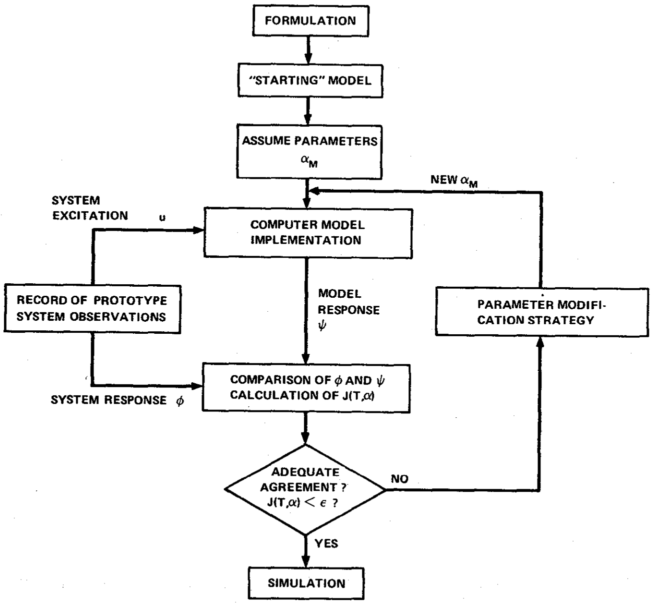

FORMULATION
"STARTING" MODEL
ASSUME PARAMETERS α_M
NEW α_M
SYSTEM EXCITATION u
COMPUTER MODEL IMPLEMENTATION
RECORD OF PROTOTYPE SYSTEM OBSERVATIONS
MODEL RESPONSE ψ
PARAMETER MODIFICATION STRATEGY
SYSTEM RESPONSE ϕ
COMPARISON OF ϕ AND ψ CALCULATION OF J(T,α)
ADEQUATE AGREEMENT ? J(T,α) < ε ?
NO
YES
SIMULATION

Figure 1—Conventional parameter identification

considerable sampling errors, and never sufficiently complete to satisfy the analyst. Usually, these data are passively-obtained, constituting responses to incompletely-known excitations over which the analyst has no control. The ultimate objective of the modelling effort is to generate a computer model which can be utilized, during the simulation phase, to investigate a variety of hypothetical control situations and which can be used to predict the response of the system to these control strategies. There are, of course, many system identification problems which do not have these characteristics. It is conceded, therefore, that the type of modelling discussed in this paper is directly applicable to only one class of a broad spectrum of modelling problems.

THE CONVENTIONAL MODELLING METHOD

The approach most often used in the construction of models of the type discussed in the preceding section involves the iterative refinement of an assumed model, by comparing the response of the model with the response of the prototype system and by modifying the model so as to minimize the difference between the two. This is illustrated in Figure 1 and discussed in considerable detail by Balakrishnan[3] and Bekey.[4] The following are the major steps in the conventional method:

1. *Formulation:* The basic governing equations and all specific physical information applying to the system under study are assembled, together with all available excitation/response data. The basic equations generally have the vector form

$$\overset{\circ}{\phi}=f(\phi,\ u,\ \alpha_s,\ t) \tag{2}$$

where ϕ is the response vector, u is the excitation, and α_s is the system parameter vector.

2. *"Starting" Model:* On the basis of all available evidence and insight, an initial hypothesis as to the model is made. This includes an initial specification of the governing equations, the structure or geometry of the system, and the system parameters. The model equations have the general vector form

$$\overset{\circ}{\Psi}=f(\Psi,\ u,\ \alpha_M,\ t) \tag{3}$$

where Ψ is the response of the model to the excitation u, and α_M is the model parameter vector.

3. *Implementation:* The equations characterizing the "starting" model are programmed on a computer. The computer model is then subjected to excitations similar to those recorded for the prototype system under study, and the response of the model to these excitations is obtained.

4. *Criterion:* A criterion function is specified to serve as a measure of the extent to which the response Ψ of the model conforms to the response ϕ of the prototype system being modelled. Usually this criterion function is defined by an expression of the type

$$J(T,\ \alpha_M)=\int_0^T\ (\phi-\Psi)'\ W\ (\phi-\Psi)\ dt \tag{4}$$

where $W = W(\phi,\ \psi,\ t)$ is a suitable weighting function, and T is the time interval over which the identification takes place. This criterion function $J(T,\ \alpha_M)$ is calculated from the system and model responses. That is, the response of the model and the response of the system are compared.

5. *Decision:* The objective of the identification procedure is to seek an optimum set of parameters $\bar{\alpha}_M$ which minimize the criterion function such that

$$J(T,\ \bar{\alpha}_M)=\min_{\alpha_M}\ J(T,\ \alpha_M) \tag{5}$$

The criterion function calculated in step 4 is, therefore, examined to see if it exceeds a specified minimum ϵ. If it does not, the identification is complete, the model is considered valid and employed for simulation. If the criterion function is not sufficiently small, the model must be modified.

6. *Modification:* A computational procedure, usually in the form of algorithms, is specified. This routine defines the manner in which the model parameters α_M are to be modified after each iteration, and it may involve gradient methods, random search, relaxation, etc. In any event, it acts to change the parameters of the model hopefully in a manner which results in a smaller $J(T, \alpha_M)$.

The conventional method of modelling is effective for the identification of systems which are "well-behaved." In particular it works well in situations in which the initial guess as to the model is very close to the prototype system, and where the excitation/response data are of very high quality. The method breaks down in many practical applications, however, for two principal reasons: 1. The first hypothetical model is not a sufficiently-close representation of a prototype system, and 2. The excitation/response data available from observations of the prototype system are of such low quality that the attaining of a minimum in the criterion function cannot be taken with confidence as an indication of the validity of the model.

To illustrate the weakness of conventional modelling, consider again the underground water resource modelling problem discussed above. Whereas Equation (1) can be assumed to apply reasonably well throughout the aquifer, the geometry of the field (boundaries of the porous medium), the initial conditions, the excitations, as well as the presence of major inhomogenieties are only incompletely known. The first hypothetical model is, therefore, likely to be substantially different from the actual system. The system response data which are to be used to improve this initial guess are represented by well-logs at haphazardly-spaced points in the field and constitute measurements of the dependent variable sampled at insufficiently-frequent intervals and subjected to serious measurement errors. Nonetheless, the conventional modelling approach requires the iterative refinement of the initial model until a criterion function of the type of Equation (4) is minimized; that is, until the transient response curves of the model are fitted closely enough to the field data. As a result, even if after much laborious and elegant computation, one arrives at a model which provides a tolerable match of the field data, there remains considerable uncertainty as to the meaningfulness of the model and its usefulness in subsequent simulations. This unfortunate consideration applies to models and simulations in a wide variety of important areas of application.

THE PATTERN RECOGNITION APPROACH

The approach to modelling suggested in the present paper is based upon the following premise: the excitation/response data available from experiments or observations of a prototype system contain a large amount of potentially-valuable and useful information which is not adequately utilized in the conventional approach to modelling. In the attempt to employ curve- or data-fitting methods to match the responses of a dubious model to highly error-prone experimental observations, many key features inherent in the experimental data are averaged out, overshadowed, or simply not utilized. A reason for this lies in the application of the criterion function, such as Equation (4), during each iterative cycle, which involves an attempt to compare the "artificial" responses of the model with the "real-world" responses of the system at each stage of the modelling process.

The pattern recognition approach to modelling is similar in some respects to that employed by Duda[5] and others in recognizing and classifying handwritten characters. In that method, the pattern recognition problem is viewed as a sequence of four mappings as shown in Figure 2. The handwritten characters themselves constitute a so-called "object space" (z). By means of a video scan of the characters, followed by sampling and digitizing, the object space is mapped into a "representation space" (y) consisting of a sequence of binary numbers. Algorithms are then developed to map from the "representation space" into a "feature space" (x). This mapping, termed feature extraction, involves ignoring most of the available samples and the focusing of attention on a few key sampled values which are sufficient to distinguish the characters from each other. Finally, there follows a mapping from the "feature space" to the "decision space" (d), a classification operation in which the features that have been extracted are used to decide the identity of a character under examination.

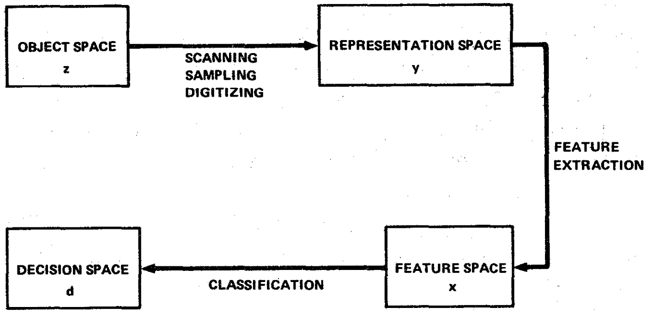


Figure 2—Successive mappings in pattern recognition

In character recognition, no attempt is made to devise a criterion function of the type of Equation (4) in order to identify characters. Rather the one-dimensional sequence of video signals is subjected to feature extraction, such that a small number of video samples are examined to determine whether they are black or white. The decision as to whether a given character is or is not the letter A, for example, is made on the basis of whether these key characters, sometimes termed the "mask," are of the correct combination of black and white. This mask is developed by postulating a "starting" mask and by working with a "learning set." The learning set is a collection of handwritten characters obtained from representative collections of manuscripts. The "starting mask" and a decision algorithm are then used to examine the learning set, and the success or failure of the character identification is recorded. The mask and the decision algorithm are then modified and applied to the same sequence of characters. This is repeated for many different masks and decision algorithms. The mask and algorithm which manifest the best record of success are adopted as the pattern recognition algorithm, and are then applied to unknown characters as required.

The pattern recognition method of modelling has the same starting point as the conventional approach. Experimental system data (input and output measurements) are assembled, and a first hypothetical model is formulated and implemented on the computer. At this point, the two approaches part company. In the pattern recognition approach, the model implemented on the computer is not regarded primarily as something to be iteratively matched to reality (the system outputs). Rather it is considered as a "learning machine" to develop feature extraction and classification algorithms which will eventually serve to extract pertinent information from the "real world" system data. The primary objective of this first stage of modelling is not to progressively refine the model, but rather to develop a set of computing routines which can subsequently be utilized to analyze the data available from the system to be modelled. The results of this analysis then are used to formulate the "starting" model for conventional parameter identification. As shown in Figure 3, the modelling problem is therefore subdivided into two stages: pattern recognition and parameter identification.

The term "pattern" is used in the present context to connote general or global characteristics of the system being modelled. For any specific modelling problem, these patterns must be known in order to talk meaningfully of parameters and their identification. Accordingly, a list of patterns is prepared, and the nature of these patterns is to be extracted from available system observations (excitations/response data). Where possible, these patterns are formulated in such a manner that their recognition involves the answering of a yes/no question. For example, in the case of a distributed system such as that described by Equation (1), these questions might include:

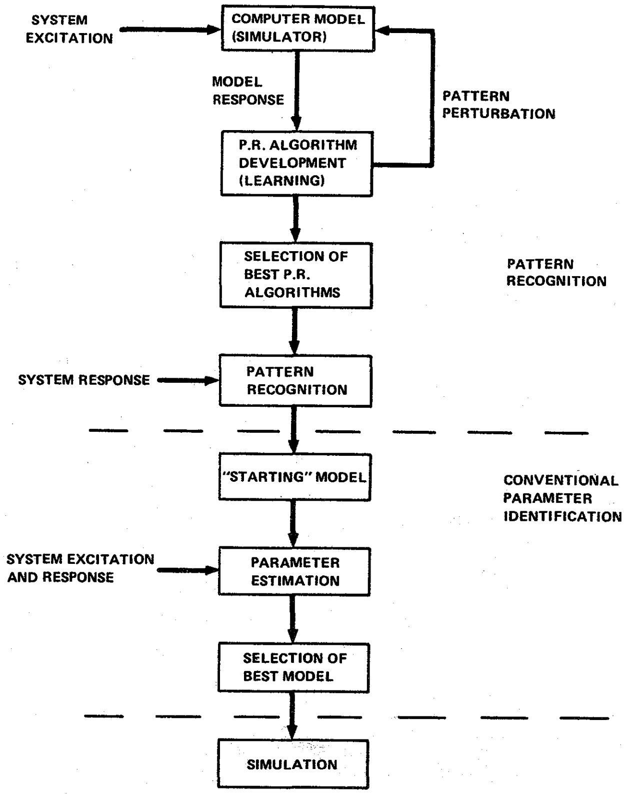


Figure 3—Steps in modelling

1. Is a given parameter (for example S) present in non-negligible quantities? That is, is it necessary to include that parameter in the model?
2. Is this parameter constant, in the range of dependent and independent variables for which system observations are available?
3. Is this parameter a function of the independent space variables, x and y?
4. Is this parameter a function of time?
5. Is this parameter a function of the dependent variable (nonlinear)?
6. Does the magnitude of this parameter everywhere fall within a specified range?
7. Considering the quality of available system observation data (number of measuring stations, sampling interval in time, and measurement

errors) is it possible to derive a model of a given dimensionality? That is, do available response data permit the meaningful construction of a finite difference grid of a specified truncation interval?

Similar questions can be asked regarding the geometry of the system, that is the location of field boundaries, and even the general structure of the basic equations. Usually in conventional modelling, all patterns of the type listed above are assumed initially, and a basic error in these assumptions invalidates all subsequent modelling efforts. In the pattern recognition method, a set of algorithms is developed with the express purpose of extracting the answers to these questions from available prototype system observations. The computer model is used to develop these computing routines.

Each algorithm is designed to accept, as its input, the response data of the model and eventually response data of the system being modelled. The model is designed to provide data having the same sampling interval, spatial distribution, and measurement noise as the original system. The output of each algorithm is the answer to one question of the type posed above. Each algorithm is, therefore, a separate pattern recognition routine. This routine can conceivably involve transformations or spectral analysis, or it may involve cross-correlations of response data taken at different points in space, but will more often take the form of a "mask." Instead of processing all the samples obtained from all response functions, attention is focused on a few key sampled values. The yes/no decision is based upon the information contained in these samples. The optimum mask, that is the combination of samples which are processed to determine whether the answer to a question is "yes" or "no," is determined experimentally using the computer model.

The pattern recognition algorithms used in modelling are developed in a manner basically similar to that used in character recognition. A "starting" algorithm is adopted either from experience or from heuristic considerations. This algorithm is tried out on the model response transients, where these response functions are generated by exciting the computer model with excitations similar to those which excited the prototype system. The algorithm also acts to "perturb" the model, so that the effectiveness of the algorithm over a number of similar yet different model configurations or parameter distributions (patterns) is determined. The algorithm is then modified automatically or by an on-line operator and the process repeated. After a number of such experiments, that algorithm which proved most effective in identifying the desired pattern is selected. The same procedure is followed to obtain successful algorithms for identifying all of the other patterns of interest, so that eventually a library of algorithms is formed—algorithms which are tailor-made for the system being modelled and for the specific excitations and responses which are available from the physical system.

Once this library is complete, attention is turned, for the first time, to the response data of the physical system. These data are now processed by the algorithms that were just developed. That is, the pattern recognition algorithms are employed to determine the patterns of the physical system. This process may demonstrate that the model used for "learning" differs radically from the system being modelled. Accordingly, this model is modified so as to give it the patterns which were found to be contained in the system being modelled. This whole process is repeated until the pattern extracted from the physical system corresponds reasonably closely to those assumed for the system being modelled. At that point, one can conclude that the model is "within the correct ball park," and the conventional parameter identification method can be employed to determine the fine structure of the model.

The general approach is illustrated in Figure 4 and takes the following steps:

1. *Formulation:* The basic governing equations and all specific information applying to the system under study are assembled, together with all available excitation/response data. The basic equations take the vector form

$$\overset{\circ}{\phi}=f(\phi, u, \alpha_s, \beta_s, t) \tag{6}$$

where β_s is a vector of patterns.

2. *"Starting" Model:* On the basis of all available evidence and insight, an initial hypothesis as to the model is made. The model equations have the general form

$$\overset{\circ}{\psi}=f(\Psi, u, \alpha_M, \beta_M, t) \tag{7}$$

where α_M is the model parameter vector and β_M is the model pattern vector.

3. *Implementation:* The equations characterizing the "starting" model are programmed on a computer. Provision is made in this implementation for perturbing or modifying the patterns of the model under control of the pattern recognition (P.R.) algorithms. The model is to accept as input data the observations of the excitation, u, of the system being modelled. The model response, ψ, is given as nearly as possible the same characteristics as the system response, ϕ. That is, response data are read out from the same

locations as those at which system response data is available, a similar sampling interval is employed, and if appropriate, noise is artificiallv added to the model output.

4. *"Starting" Pattern Recognition Algorithm:* On the basis of previous experience and insight, a separate algorithm is provided for each of the patterns, β_M to be recognized. These algorithms may include masks for selecting key samples for further processing.

5. *P.R. Algorithm Implementation:* The "starting" algorithms are programmed on the computer. These algorithms may contain loops which act to perturb or modify the patterns of the computer model so as to test the algorithms under a number of different situations. For example, if the purpose of the algorithm is to determine whether a given parameter is constant or not, that parameter is given a number of different constant values as well as caused to vary in a prescribed fashion. The modified patterns imposed by the P.R. algorithm are denoted by $\beta_M.^*$

6. *P.R. of Model Response:* The algorithm is employed to process the model response, ψ, and to recognize the model patterns for each of the model perturbations. The patterns recognized by the algorithm are denoted by $\hat{\beta}_M.^*$

7. *Comparison:* For each pattern recognition run, the success or failure of the algorithm is determined by comparing the pattern of the model, $\beta_M{}^*$ with that determined by the algorithm, $\hat{\beta}_M.^*$

8. *Criterion:* A figure of merit for each algorithm is determined by totaling the successes and failures of the algorithm over all the experiments conducted with that algorithm.

9. *Decision:* A decision is made as to whether or not additional modifications of the P.R. algorithms should be attempted.

10. *Algorithm Modification:* Either automatically or with the aid of an on-line operator, the P.R. algorithm is modified. This modification may involve the re-specification of the mask, a change in the manner in which the samples are processed, or it may involve a more fundamental change in strategy. Evidently, the specific nature of this modification depends upon the patterns to be recognized by the algorithms. In any event, steps 5 through 9 are repeated until no additional modifications are required.

11. *Selection:* Provided no additional algorithm modifications are required, that algorithm having the best percentage of success is selected and stored.

12. *P.R. of System Response:* The selected algorithms are now employed to process the system response, ϕ, obtained from prototype system observations. That is, the algorithms are employed to recognize the patterns, β_s, in ϕ.

13. *Comparison:* The patterns β_s recognized using the system observations are compared with the patterns β_M initially assumed for the model. That is, it is verified whether the model used for algorithm development was "in the correct ball park."

14. *Decision:* The results of the comparison of all the members of the pattern vectors β_M and β_s are analyzed to determine whether the "starting" model was close enough to the system being observed. If agreement between the two is adequate, that is, if the model has most or all of

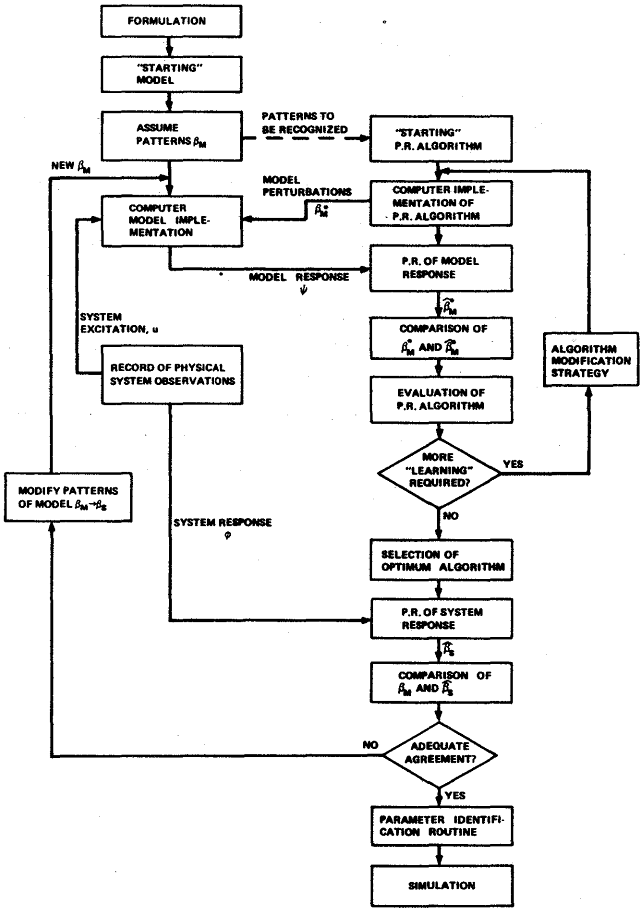

FORMULATION
"STARTING" MODEL
ASSUME PATTERNS β_M
PATTERNS TO BE RECOGNIZED
"STARTING" P.R. ALGORITHM
NEW β_M
MODEL PERTURBATIONS
β_M^*
COMPUTER MODEL IMPLEMENTATION
COMPUTER IMPLEMENTATION OF P.R. ALGORITHM
MODEL RESPONSE ψ
P.R. OF MODEL RESPONSE
$\hat{\beta}_M^*$
SYSTEM EXCITATION, u
COMPARISON OF β_M^* AND $\hat{\beta}_M^*$
ALGORITHM MODIFICATION STRATEGY
RECORD OF PHYSICAL SYSTEM OBSERVATIONS
EVALUATION OF P.R. ALGORITHM
MORE "LEARNING" REQUIRED?
YES
NO
MODIFY PATTERNS OF MODEL $\beta_M \rightarrow \beta_s$
SYSTEM RESPONSE φ
SELECTION OF OPTIMUM ALGORITHM
P.R. OF SYSTEM RESPONSE
$\hat{\beta}_s$
COMPARISON OF β_M AND $\hat{\beta}_s$
NO
ADEQUATE AGREEMENT?
YES
PARAMETER IDENTIFICATION ROUTINE
SIMULATION

Figure 4—The Pattern Recognition (P.R.) modelling method

the patterns of the physical system, the pattern recognization process is considered complete, and the computer model can be employed as the starting point for conventional parameter identification and eventually for simulation.

15. *Model Modification:* If agreement between the model and the physical system is inadequate, the computer model is modified by giving it the patterns determined in step 12. Steps 5 to 14 are then repeated until adequate agreement is obtained.

The most difficult steps in this method are the selection of the "starting" algorithm for pattern recognition and classification and the specification of the modification strategy of this algorithm. These depend strongly upon the type of patterns to be recognized, upon the computer model, and upon the nature of the response data. It is necessary, therefore, to build up a considerable amount of experience with this method for any specific application area. Occasionally it may turn out that a proven algorithm modification strategy does not lead to adequate convergence for a specific problem. This may then be taken as an indication that the quality of available response data is insufficient to permit meaningful pattern recognition. For example, the time sampling-interval may be too large, or response data may not be available for enough points in the space domain, or the signal-to-noise ratio may be too low. Under these circumstances, the computer model and the pattern recognition method can be employed to determine the approximate extent to which system observation data must be improved to make modelling possible. This can be accomplished by gradually improving the quality of the computer model response (by sampling it more frequently, for example) until the algorithm modification strategy leads to successful convergence. The results of this computer experiment are then used as the basis for better and more complete field measurements.

CONCLUSIONS

The pattern recognition method described in this paper is evidently not a panacea. The procedure is useful only for the identification of systems of "a certain shade of gray," and it leans heavily upon the ingenuity and insight of the analyst. It does, however, constitute a novel utilization of computer models—the development of a "learning set" and the determination as to whether the system response data are of sufficient quality to permit parameter identification. The approach has been used with some success in the modelling of underground water reservoirs of the type characterized by Equation (1) as well as in the study of aquifer pollution problems. The results of these studies will be reported in separate papers.

REFERENCES

1 W J KARPLUS V VEMURI
Heuristic optimization and identification in hybrid field simulations
Proc Fifth Int Congress of AICA Lausanne Switzerland pp 345-350 1967

2 V VEMURI W J KARPLUS
Identification of nonlinear parameters of ground water basins by hybrid computation
Water Resources Research Vol 5 pp 172-185 1969

3 A V BALAKRISHNAN V PETERKA
Identification in automatic control systems
Automatica Vol 5 pp 817-829 Pergamon Press 1969

4 G A BEKEY
System identification—an introduction and a survey
Simulation Vol 15 pp 151-166 1970

5 R O DUDA
Elements of pattern recognition
Adaptive Learning and Pattern Recognition Systems (J M MENDEL and K S FU, editors) Academic Press pp 3-33 1970

Horizontal domain partitioning of the Navy atmospheric primitive equation prediction model

by E. MORENOFF

Ocean Data Systems, Inc.
Rockville, Maryland

and

P. G. KESEL and L. C. CLARKE

Fleet Numerical Weather Central
Monterey, California

INTRODUCTION

Development of the Kesel-Winninghoff multi-layer baroclinic primitive equation atmospheric prediction model began at the Fleet Numerical Weather Central, Monterey, California, in late 1968. The model, herein referred to as the Primitive Equation Model (PEM), was initially written as a single processor version to be executed in one of the dual processors of one of the two FNWC CDC 6500 dual processor computer systems. This version, however, required slightly over six and one-half hours to compute a set of 72 hour predictions.

Prior to its employment on an operational basis in September 1970, a four processor version of the PEM was developed which produced the same results as the single processor version in significantly less elapsed time. In particular, the PEM was partitioned to take advantage of all possible computational parallelism to exploit the four powerful central processing units available in the FNWC computer installation. The execution of the PEM for a 72 hour prediction run was thereby reduced from 405 minutes in the one-processor version to 135 minutes in the four-processor configuration.

A description of the partitioning of the PEM on the basis of the equation partition, the mode of operation of that version of the PEM, and the results attained through its employment were presented in earlier papers.[1,2] At the conclusion of the first paper, plans were revealed for the development of a second multiprocessor version of the PEM, one in which the partitioning was to be based on the horizontal domain rather than the equation set. That effort has now been completed and a new version of the PEM, partitioned according to horizontal grid space considerations, has been operational at FNWC since October 1971, with significant improvements in terms of both elapsed time and central memory size requirements to generate the 72-hour forecast.

This paper summarizes the principal factors involved in the repartitioning of the PEM. First, the PEM, and the mechanisms by which the partitions of the PEM in each of the four processors communicate with one another and their executions are synchronized, are reviewed. Next, the PEM structure and its partitioning are described. Finally, the results of the reduction to operational usage and future developmental efforts are presented.

THE PRIMITIVE EQUATION MODEL

The current version of the PEM better models the physical processes than the previous versions. Principal characteristics of the model are reviewed herein, including those which led to the improvement in the forecast skill of the PEM.

The governing equations of the PEM shown in Figure 1 are written in flux form in a manner similar to Smagorinsky et al.,[3] Arakawa,[4] Arakawa, Katayama, and Mintz,[5] and Langlois and Kwok.[6] The corresponding finite difference equations (which are not shown) are based on the Arakawa conservation technique. This scheme precludes nonlinear instability by requiring

$$\frac{\partial(\pi u)}{\partial t} = -m^2 \{\frac{\partial}{\partial x} (\frac{uu\pi}{m}) + \frac{\partial}{\partial y} (\frac{uv\pi}{m})\} + \frac{\partial}{\partial \sigma} (wu) + \pi vf - m(\pi \frac{\partial \phi}{\partial x} + RT \frac{\partial \pi}{\partial x}) + D(u) + F(u)$$

$$\frac{\partial(\pi v)}{\partial t} = -m^2 \{\frac{\partial}{\partial x} (\frac{uv\pi}{m}) + \frac{\partial}{\partial y} (\frac{vv\pi}{m})\} + \pi \frac{\partial}{\partial \sigma} (wv) - \pi uf - m(\pi \frac{\partial \phi}{\partial y} + RT \frac{\partial \pi}{\partial y}) + D(v) + F(v)$$

$$\frac{\partial(\pi T)}{\partial t} = -m^2 \{\frac{\partial}{\partial x} (\frac{\pi uT}{m}) + \frac{\partial}{\partial y} (\frac{\pi vT}{m})\} + \pi \frac{\partial}{\partial \sigma} (wT) + \frac{RT}{C_p \sigma} \omega + D(T) + H(T)$$

$$\frac{\partial(\pi q)}{\partial t} = -m^2 \{\frac{\partial}{\partial x} (\frac{\pi uq}{m}) + \frac{\partial}{\partial y} (\frac{\pi vq}{m})\} + \pi \frac{\partial (wq)}{\partial \sigma} + D(q) + Q(q)$$

$$\frac{\partial \pi}{\partial t} = - \int_0^1 [m^2 \{\frac{\partial}{\partial x} (\frac{u\pi}{m}) + \frac{\partial}{\partial y} (\frac{v\pi}{m})\} + \pi \frac{\partial w}{\partial \sigma}] \, d\sigma$$

$$\frac{\partial \phi}{\partial \sigma} = - \frac{RT}{\sigma}$$

$$\pi = \rho \frac{RT}{\sigma} \qquad \text{where } \sigma \equiv P/\pi \text{ and } w \equiv -\dot{\sigma}$$

D( ) = lateral diffusion operator
F( ) = surface friction operator

Q( ) = moisture source and sink terms
H( ) = diabatic heating terms

Figure 1—The equation set

that the flux terms conserve the first and second moments of any advected parameter, assuming continuous time derivatives. Total energy is conserved because of constraints placed upon the vertical differencing. Total mass is conserved when integrated over the entire domain. Linear computational instability is avoided by meeting the Courant-Friedrichs-Lewy criterion.

The Phillips[7] sigma vertical coordinate is employed in which pressure is normalized with the underlying terrain-level pressure. At levels where sigma equals 0.9, 0.7, 0.5, 0.3, and 0.1, the horizontal wind components, *u* and *v*, the temperature, *T*, and the height, *z*, are carried. See Figure 2. The moisture variable, *q*, is carried at the three lowest levels. Vertical velocities, *w* (defined as minus sigma dot), are carried at the layer interfaces (sigma equals 0.8, 0.6, 0.4, 0.2) and are calculated diagnostically from the vertically-integrated continuity equation. At the top and bottom of the column the vertical velocities vanish identically.

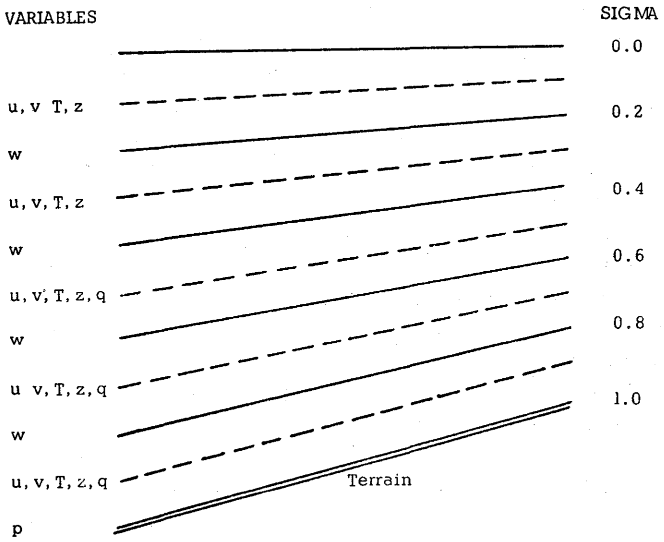


Figure 2—Diagram of levels and variables

The Clarke-Berkofsky mountains are used in conjunction with both a Kurihara[8] form of the pressure-force terms in the momentum equations, and with slight amounts of lateral diffusion to eliminate the customary "noise" patterns over high, irregular terrain.

The Richtmyer centered time-differencing method is used with a ten-minute time step, but integrations are recycled every six hours with a Matsuno (Euler backward) step to reduce solution separation.

The earth is mapped onto a polar stereographic projection of the Northern Hemisphere. The grid lattice has 63 rows and columns, and the geographical equator is an inscribed circle. The mesh length is 381 kilometers at 60 degrees North (and about one half this distance in the extreme corners of the array).

A considerable part of the diabatic heating and moisture terms in the model was based on the work of Mintz and Arakawa as described by Langlois and Kwok. Climatological values of the earth's albedo are used. A Smagorinsky parameterization of cloudiness based on layered relative humidities is used in the radiative flux calculations. Dry convective adjustment precludes hydrostatic instability. Moisture and heat are redistributed in the lowest three layers by use of an Arakawa parameterization of three types of cumulus cloud ensembles. Convective precipitation is permitted in two of these three cloud types. Evaporation and large-scale (cyclones) condensation are important source-sink terms in the moisture conservation equation. Evaporation over land, however, is based on a Bowen ratio, using data from Budyko. In the calculation of sensible heat fluxes over water, the FNWC-produced sea surface temperature distribution is invariant with time but updated for each forecast. Over land, the surface temperature is obtained from a heat balance equation. Surface stress is computed for the lowest layer.

The type of lateral boundary conditions which led to best over-all results was the constant flux, restoration boundary conditions devised by Kesel and Winninghoff, and implemented in 1970. The procedure is as follows: A field (63×63 array) of restoration coefficients which vary smoothly from unity at and south of 7.5 degrees North to zero at and north of 15 degrees North is computed once and saved. At the conclusion of each ten-minute integration step the (new) values of the state variables are restored back toward their values at the previous time step according to the amount specified by the restoration coefficient at each grid point. The net effect of this technique is to produce a fully dynamic forecast north of 15 North, a persistence forecast south of 7.5 North, and a blend in between. The blend region acts as an energy sponge for outwardly propagating inertia-gravity waves.

The basic inputs for the model are the virtual temperature and height distributions for the Northern Hemisphere at twelve constant pressure levels between the surface and 50 MBS, moisture distributions at four levels between the surface and 400 MBS, the sea surface temperature distribution, the sea level pressure distribution. These analyses are generated twice daily on an operational basis, and are derived from about 550 upper air reports (temperature, pressure, moisture, winds) and 4,500 sea level observations. These reports are augmented by aircraft observations (mainly between 30,000 and 40,000 feet) in large numbers, and satellite soundings (SIRS data) sporadically.

INTER-PROCESSOR COMMUNICATIONS AND SYNCHRONIZATION

The inter-processor communications and synchronization mechanisms are identical to those employed in the version of the PEM partitioned on the basis of the equation set as reported in the previously referenced paper. These mechanisms are briefly reviewed in this section for purposes of clarity.

The two FNWC dual-processor CDC 6500 computer systems can be linked with each other through the one million words of Extended Core Storage (ECS) operated in a mode such that the entire ECS is accessible by either CDC 6500 computer system. When the PEM is to be executed, the four programmed partitions of which it is comprised are assigned to and loaded in each of the four processors. One of the program partitions is designated as the master partition and the remaining three as the slave partitions by the use of appropriate ECS access codes and pass keys. If the ECS access code field indicates the partition to be the master, the associated pass key is interpreted as the name of the ECS block storage assigned to the PEM. The slave partitions have no ECS assigned to them but are able to refer to the same ECS block as the master partition by use of the same pass key.

Communications between the program partitions being executed in the different CDC 6500 computer systems are established through the aid of a FNWC developed Peripheral Processor (PP) routine, 1SI, which links the two operating systems in each of the two computers. Hence, 1SI provides a software, full duplex block multi-plexing channel between the two computers via ECS. Messages and/or blocks of data may be sent over this channel so that 1SI may be used to call PP programs in the other computer or to pass data such as tables or files between the computers.

Immediately following the initiation of the execution of the four program partitions in their respective processors, the operation of the three slave partitions is delayed until the master partition requests and has been assigned the necessary ECS block storage required by the PEM, and the synchronization mechanism is initialized. ECS block storage is requested by the master partition in the same manner as any conventional job to be executed in the computer.

Once obtained, the master partition labels the ECS block storage by passing an argument comprised of an access code specifying its status as master and the desired pass key to the peripheral processor routine, ECS. The routine ECS searches the resident control point exchange area (CPEA) and, through 1SI, that of the other computer, for a master with the same pass key. If one is found, the requesting program partition is aborted. If the other computer is inoperative or if no matching key is found, the label is established.

When the operation of the three slave partitions in their respective processors is manually reinitiated following the successful assignment of ECS block storage to the master partition and the initialization of the synchronization mechanism, each slave partition passes the argument comprised of its access code indicating it to be a slave and pass key to the peripheral processor routine, ECS. This time, ECS searches its own computer's CPEA for a master with a matching key. If none is found, the search is repeated in the other computer's CPEA via 1SI. If still none is found, this fact is indicated to the requesting partition. If a match should exist in either computer, the original ECS will contain the address (ECRA) and field links (ECFL) of the requesting partition stored in its CPEA and will be given the ECRA and ECFL of the matching master partition.

The mechanism by which the parallel execution of the multiple partitions in each of the processors are exactly synchronized is based on a general program linkage mechanism known as the Buffer File Mode of Operation.[9,10,11] The application of the Buffer File Mode of Operation to the FNWC multiple computer environment requires the Buffer Files to reside in a random access storage device jointly accessible by each of the computers. The ECS satisfies this requirement when operated in the manner described above.

The nature of the information passed between any pair of partitions is whether or not one partition has reached a point in its execution where sufficient data has been developed to allow the other partition to initiate or continue its own execution. This is represented as a single "GO—NO GO" flag to be sensed by the second partition. Hence, the recirculating ring structure normally associated with the Buffer File Mode of Operation reduces to a simple single one word block maintained in ECS.

Finally, access to a Buffer File by a partition must be unidirectional. Any single Buffer File may only be *written to* by one partition and *read from* by another partition. Consequently, a pair of Buffer Files is assigned between any two partitions whose operation is to be synchronized.

PEM STRUCTURE

Each of the four partitions of which the PEM is comprised are identically structured. Each partition is considered in three distinct phases: the initialization phase; the integration phase; and the output phase. Whereas the computations associated with the integration phase are identical in each of the four partitions, those associated with the initialization and output phases vary from partition to partition.

The phases are structured as separate overlays within each partition. The relationships which exist among the overlays of a particular partition and with the overlays in the other three partitions are illustrated in Figure 3, which is a representation of the master overlays associated with each of the four partitions. The principal functions of the master overlays are to synchronize the calls for the execution of the overlays within the respective partitions with those in the other partitions, to dynamically adjust the field lengths of core storage required by the overlay to be called, and finally, to invoke the execution of the appropriate overlay.

Overlay call synchronization among the different partitions is realized via requests by the master overlays in each partition to "set" a Buffer File (SCOMM) or to "read" a Buffer File (RCOMM). A five character Buffer File naming convention was established to facilitate identification of which partitions were involved. The first two characters of the name serve to identify the Buffer File as being interstep ("IS"). The third character specifies whether a split ("S") or a join ("J") is being signaled. The fourth and fifth characters specify the partitions writing and reading the Buffer File. Hence, Buffer File ISS14 is used by partition 1 to split its operation by initiating execution of partition 4 in going from one time step to another.

The initialization phase is invoked only once per 72 hour forecast period. The integration phase is repeated in each forecast time step. Each thirty-sixth time step, the results of the preceding forecast hours are output and the integrations reiterated. The program loops extending from the DO statement through the

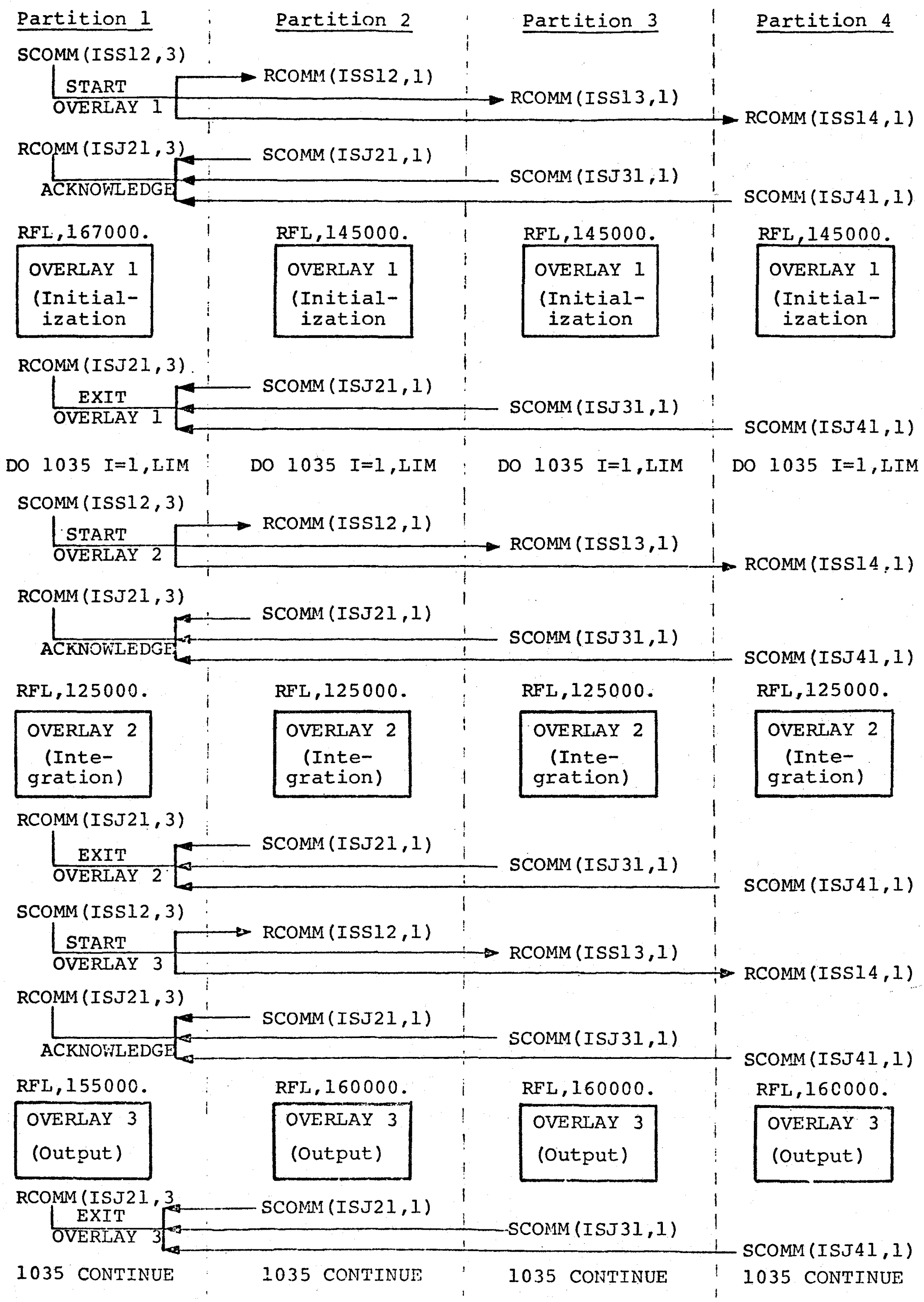


Figure 3—Partition overlay structure

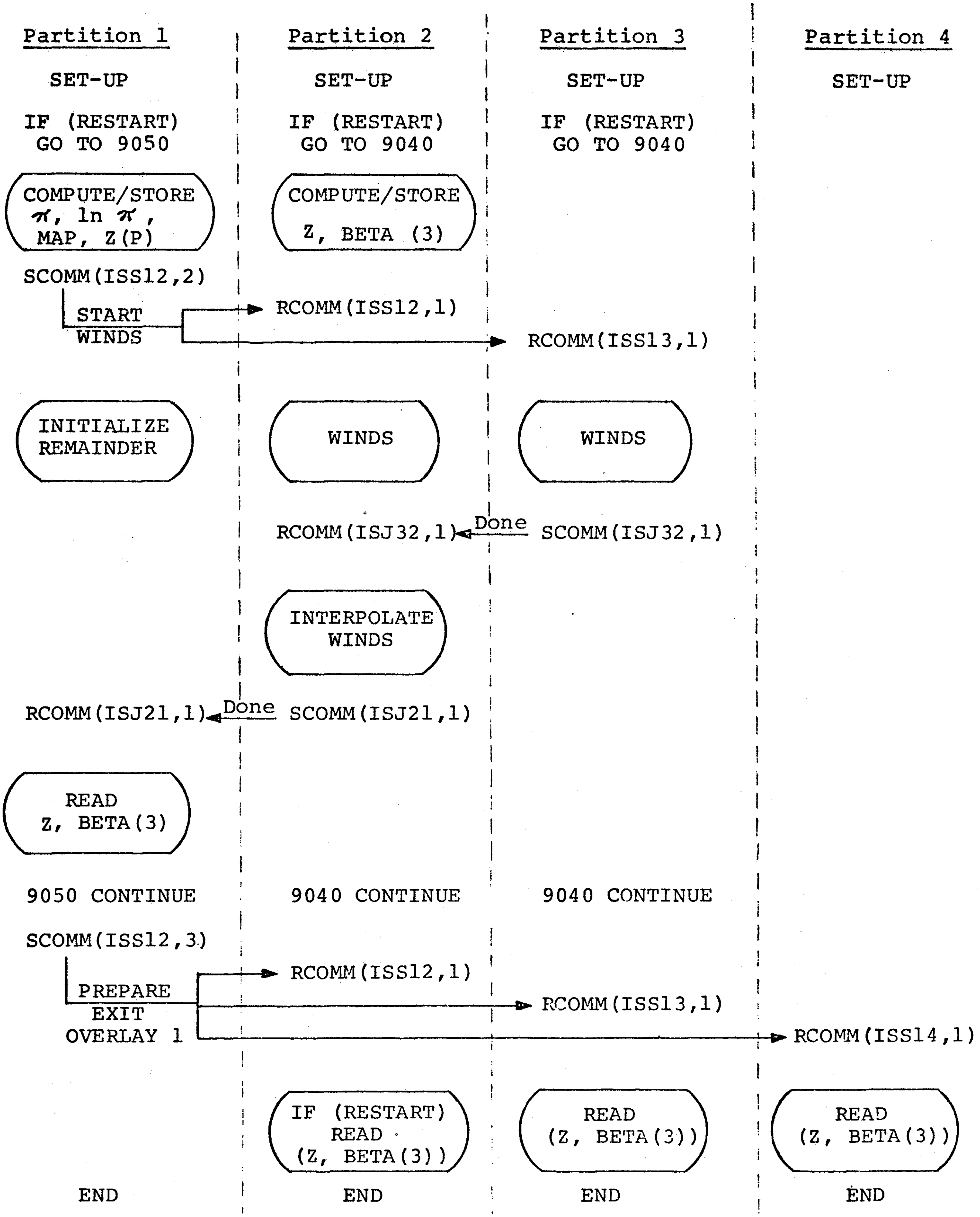


Figure 4—Overlay 1 partition synchronization

CONTINUE statement in each of the partitions of Figure 3 control the execution of the integration and output overlays. Note the variable upper limit of the number of executions of these program loops. This is manually set. The operation of the PEM may be suspended and reinitiated from the point following the completion of the most recent execution of the output overlav.

Finally, the RFL (REQUEST FIELD LENGTH) statements appearing in Figure 3 clearly indicate the variability of the main storage requirements of the overlays within each partition of the PEM. This variability is principally a result of the manner in which the overlays operate on the data fields present. The initialization overlays treat the data fields on a "full-field" basis. The integration overlays, however, treat the data fields on a "quarter-field" basis. Lastly, the output overlays treat the data fields on both a full-field and third-field basis. The utilization of the data fields by the overlays will be elaborated on in the following section. The point to be made here is that the main storage requirements of the PEM are dynamically adjusted in the course of its execution to maximize the storage available to other programs which may be concurrently sharing the FNWC computer resources with the PEM.

PEM PARTITION OVERLAYS

In addition to the synchronization of the operation of the partitions of the PEM at the master overlay level, further synchronization is required among the subordinate initialization, integration and output overlays. This additional level of synchronization is realized by the same Buffer File mechanism employed at the master overlay level.

Consider first the initialization overlays. As noted in the preceding section, the initialization overlays treat the data fields on a full-field basis and hence the partitioning of these overlays is based on computational functions rather than spatial considerations. Intra-overlay synchronization is consequently needed to insure the completion of each section of the initialization process in the appropriate partitions before the next section is allowed to be initiated. Reference to Figure 4, for example, shows that the interpolation of the winds in the initialization process in partition 2 must wait for a confirmation that the initial wind computations in partitions 2 and 3 have each been completed.

Figure 4 also illustrates the manner in which the PEM restart capability functions. In the event the operation of the PEM is being restarted, as manually noted by the operator, the computations within the initialization overlay are essentially completely bypassed. In such a case the initialization overlay is used to pass those parameters needed for the continued operation of the PEM from partition 1 to the other partitions.

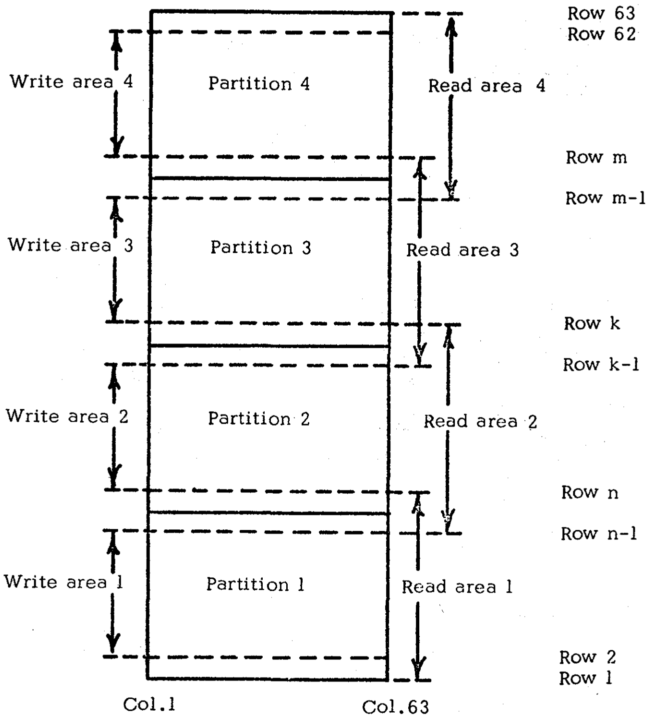


Figure 5—Horizontal doman partition

During initialization, each partition operates on "full" fields (that is, on the complete 63×63 arrays). In the integration phase each partition operates on quarter fields. During the output phase both full fields and third fields are used, depending on the operations being performed. In the output phase, for example, transformation of coordinates (from sigma surfaces to pressure surfaces) are carried out in three processors (on third fields) while the fourth processor writes sixty checkpoint restart data fields on the restart tape. Once done, all four processors can then perform full-field filtering and/or smoothing operations on one-fourth of the number of forecast fields which are written on the disk (for transmission to users).

In the integration phase, it is important to note that several alternate configurations were considered with respect to how the 63×63 data arrays could be most effectively partitioned. For example, the four-way

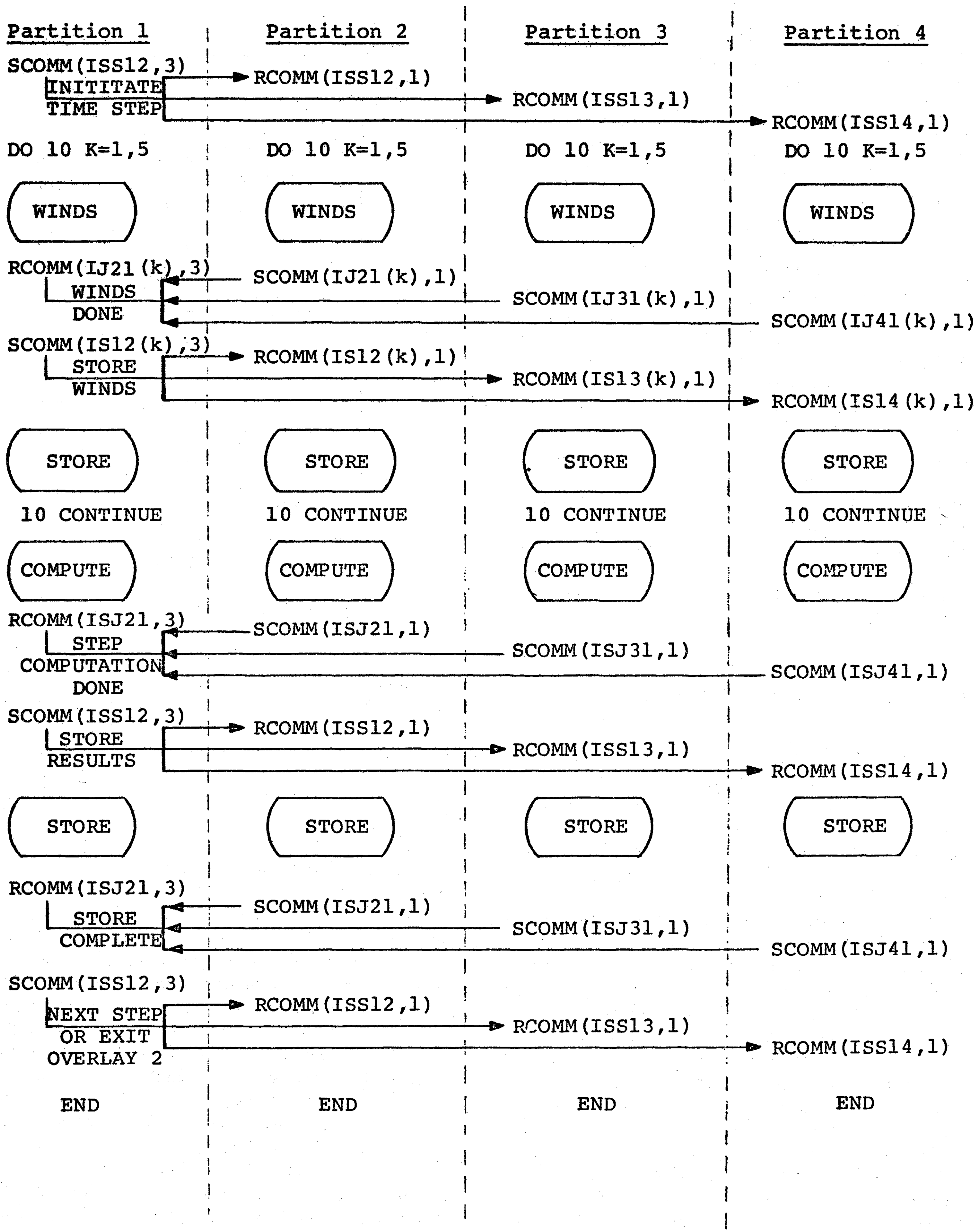


Figure 6—Overlay 2 partition synchronization

partition based on *quadrants* was rejected because an array transformation would have been required to reassemble the quarter fields into contiguous ECS locations. A detailed analysis indicated that partitioning the grid into four adjacent *rectangular strips* was the best possible scheme. See Figure 5. Using the partition method shown, nonoverlapping writes to ECS, extra-row reads from ECS for space differentiation, and time synchronization of partitions contribute both to economy and solution (no internal boundary problems).

Assume that each partition is to calculate answers for n rows in the $4n \times m$ total field. Now, because of the need to compute horizontal gradients, it was necessary to read into central memory of each partition and to calculate the quantities to be differentiated on $(n+2)$ rows (using second-order space differencing). The nonoverlapping n-rows from each partition are reassembled into total fields on ECS at the conclusion of each computational segment. This meant that the read/write first/last word addresses for data transfers to/from ECS were unique to each partition.

One final consideration needs to be covered. If all grid points in the total array were of the same class (computationally speaking), each partition should calculate on exactly one-fourth of the number of rows in the total horizontal space. But, in this particular model, three classes of grid points exist. South of 7.5 degrees North, only restoration of former parameter values takes place. North of 15 degrees North, however, the model simulates more of the physical processes than in the region between 7.5 and 15 North (where it is only diabatic). By expressing all of the addresses and field lengths in terms of easily changeable variables, it was possible to let the computer determine the optimum number of rows to be calculated in each processor. In the computations of a typical integration time step the results of the preceding time step are transferred from permanent storage to temporary working storage associated only with the particular partition. Following the computations within the time step of each partition, the results are transferred back to permanent storage.

It is important to note the number of strips into which the grid was divided was solely based on the number of Central Processing Units available in which to process the partitions. In the event, for example, ten or one hundred Central Processing Units were available, then the number of strips could have been selected as either ten or one hundred, respectively, and the number of partitions extended appropriately.

Synchronization points within the integration overlay among the four partitions of the PEM are illustrated in Figure 6. The following observations are relevant.

First, the wind computations are repeated within a DO Loop present in each partition five times, once for each level of the atmosphere. In order to insure the PEM remains in synchronization in the event the execution of one of the partitions is temporarily interrupted or suspended, a different pair of Buffer File communication cells is required for each execution of the DO Loop.

Second, the oval COMPUTE box incorporates all the remaining computations associated with the time step. Each sixth time step these computations are modified to take into consideration the effects of diabatic heating. This includes incoming solar radiation, out-going terrestrial radiation, sensible heat exchange at the air-earth interface, and evaporation. Each step the computations are further modified to take into account condensation processes.

And third, synchronization controls are provided to insure completion of all computations at a time step prior to allowing the results of that time step to be transferred to permanent storage by any of the partitions. Similarly, controls are provided to insure the completion of the transfer of the results of the time step computation to permanent storage prior to allowing the next time step to be initiated in the integration overlays of any of the partitions.

The output overlay is entered every thirty-sixth time step or sixth hour during the forecast period. The output overlay in partition 1 is devoted to duplicating onto magnetic tape from their permanent storage all data fields required to restart the PEM. At the same time, the output overlays in partitions 2, 3 and 4 are post-processing the output fields, that is, transforming coordinates, filling in values under terrain, filtering and outputting the resultant fields. The time required to prepare the restart tape is a small fraction of the time required to output the results of the previous thirty-six time steps and hence this part of the restart procedure does not appreciably extend the model's execution time.

The synchronization of the execution of the output overlays in each of the four partitions is shown in Figure 7. The routines contained in the square boxes labelled OUTPE1 are amplified in Figure 8. Note that all preprocessing of data must be completed before OUTPE1 can be invoked in any of the partitions. Further note that in OUTPE1 partition 3 assumes the role of master partition with respect to the operation of partitions 2, 3 and 4. The computations within OUTPE1 are performed on a third-field basis, one-third of the data fields being processed in each of partitions 2, 3 and 4. OUTPE1 prepares data fields for output five separate times during its execution and then calls on the routine OUTPE2(k) where $k=1, 2, 3, 4, 5$ to actually output the data fields. Synchronization controls are provided

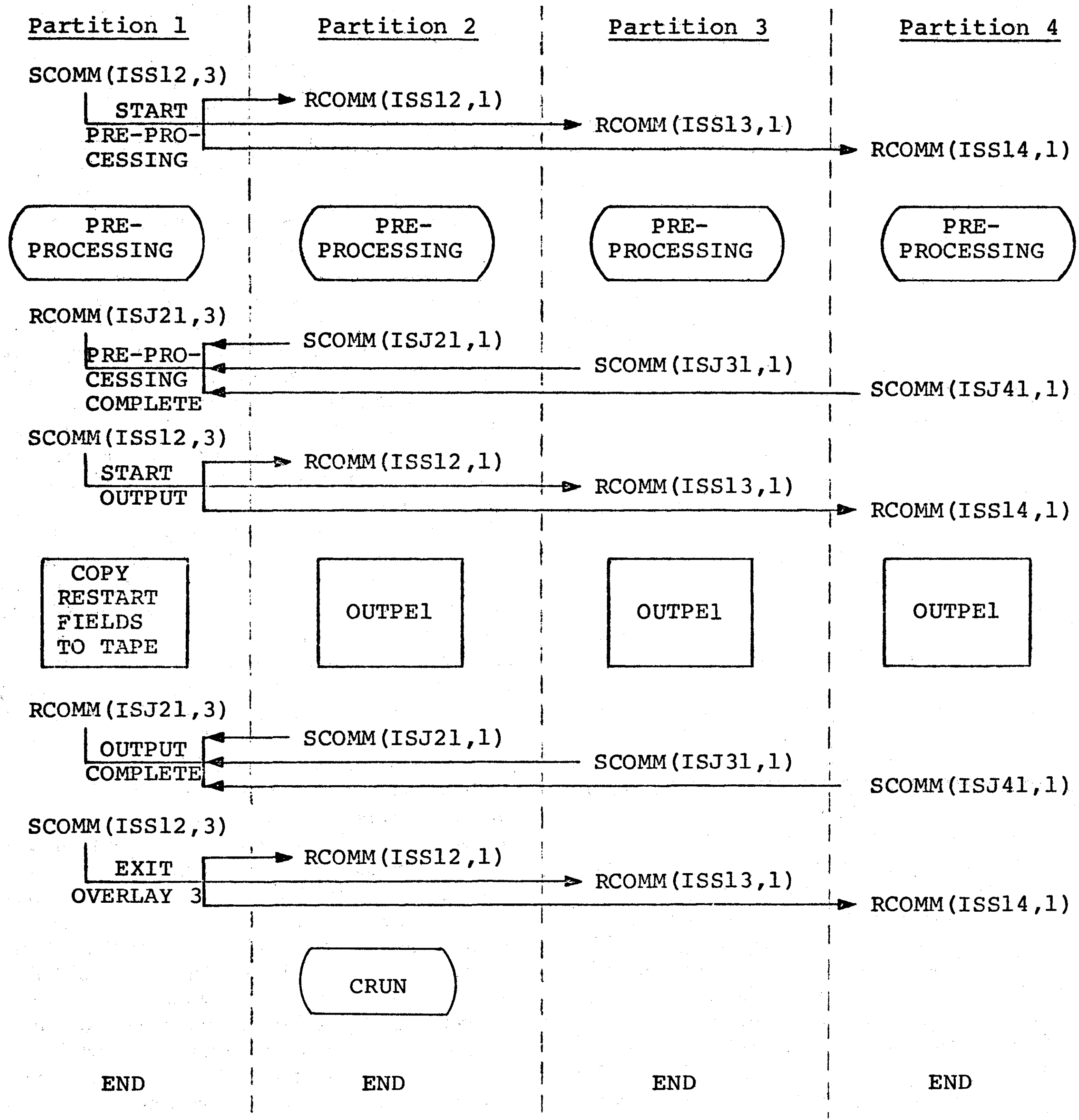


Figure 7—Overlay 3 partition synchronization

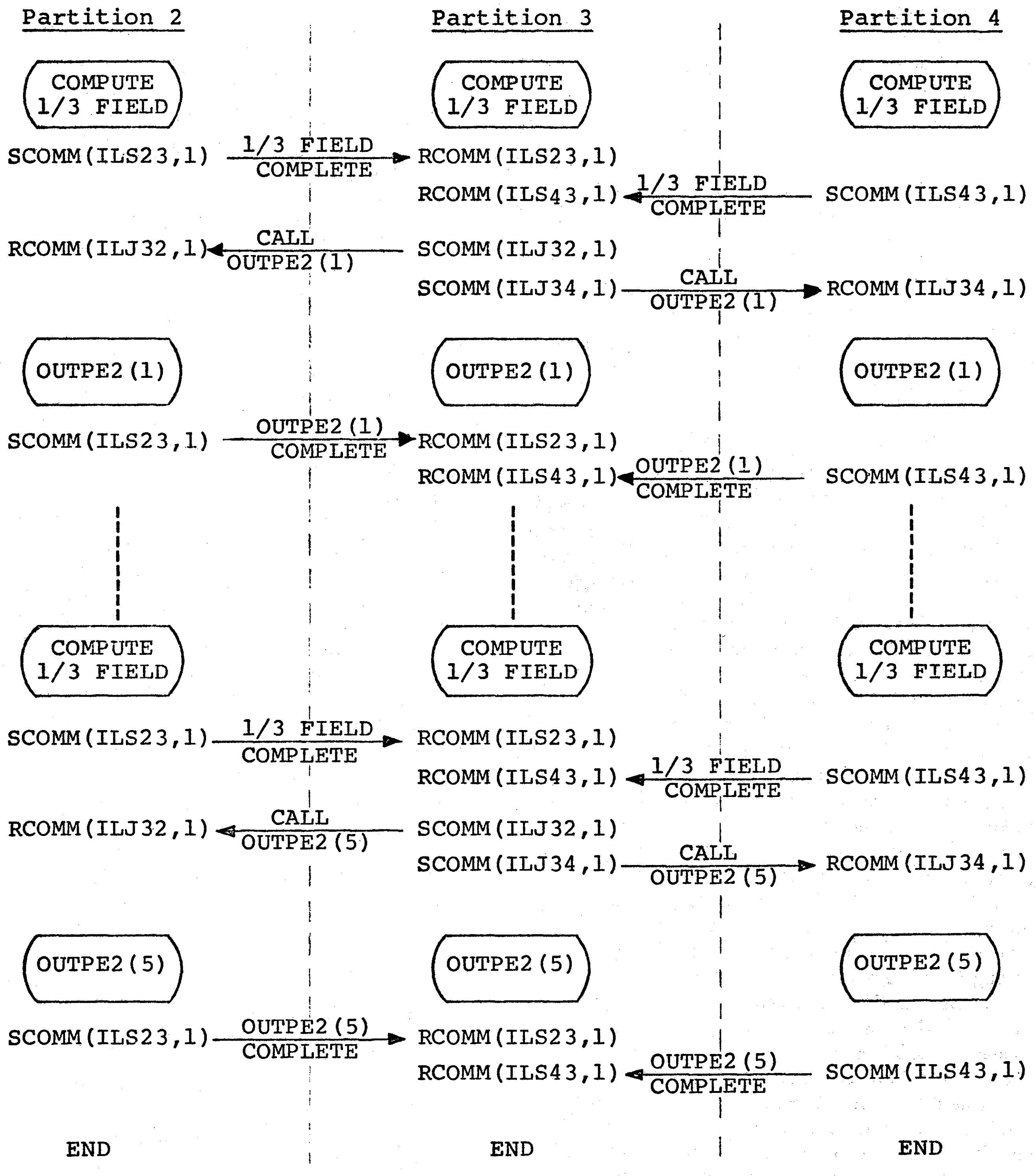


Figure 8—OUTPE1 partition synchronization

	Resolution	Number of Points	Effective* Space Increment	Composite Factor**
A.	5°/5 Layers	2450	300	0.41
B.	2.5°/5 Layers	10082	150	3.38
C.	2.5°/10 Layers	10082	150	6.76
D.	1.25°/10 Layers	40898	75	55.03
E.	1.25°/20 Layers	40898	75	110.06

* Assumes some technique to artificially eliminate over-specification at high latitudes.

** Compared to the FNWC PEM.

Figure 9—Global grid model hierarchy

to isolate the output data preparation computations and the OUTPE2(k) calls within each partition.

CONCLUSIONS

The FNWC (Kesel-Winninghoff) Primitive Equation Atmospheric Prediction Model was repartitioned on the basis of horizontal grid space rather than equation partition considerations. Although the current version of the PEM has been partitioned to take advantage of the four processors of the FNWC two dual-processor CDC 6500 computer systems, the partitioning may be directly extended in the event additional processors are made available. Hence the current version of the PEM is ideally suited for operation on parallel processor computers such as the ILLIAC IV or the CDC 8600.

As a consequence of employing the four-processor version of the PEM partitioned on the basis of horizontal domain rather than computational burden considerations, the same 72 hour meteorological forecast products were generated in 80 minutes rather than 135 minutes. In addition, the main core storage requirements of the current model are significantly less than that of the earlier version. This is due, in part, to the introduction of an overlay structure in the current model and, in part, to the performance of computations during the integration overlay on a quarter-field basis.

The current PEM has demonstrated a remarkable increase in forecast skill over the previous operational model. It models more of the physical processes better than ever before. But error analyses reveal that the forecasts still deteriorate rapidly in the smaller scales of motion being simulated because of *spatial truncation.* Spatial truncation can cause undermovement of some small-scale pressure systems by as much as twenty-five percent of the observed displacement. Another significant source of error is the *data base* itself. In spite of the receipt of over 500 upper-air soundings every twelve hours and 4,000 surface observations every six hours, the data are too sparse over oceans and aloft to correctly specify the initial conditions. With the expected proliferation of satellite probes of the atmosphere, this may not only minimize the initialization problem but also justify high-resolution global models for operational forecasting.

A hierarchy of models of varying resolution and the associated computational burden that must be overcome have been considered.[12] The composite computation factor is normalized in terms of the size of the PEM problem being solved today on two CDC 6500 computers. See Figure 9.

Recall that the current PEM has the following attributes: five layers, 4,000 grid points per layer, hemispheric, 200 nautical mile mesh (at 60 degrees North), and ten-minute time steps. Figure 9 shows that latitude-longitude grids of increasing resolution (both horizontal and vertical) could lead to problems requiring two orders of magnitude more computations than are currently being done operationally without any serious risk of over-specification (assuming large quantities of satellite soundings).

If one assumes a fifty percent efficiency for a computer of the ILLIAC IV class, it might be possible to obtain about 500 Million Instructions Per Second (MIPS). FNWC's two CDC 6500 Computing Systems generate about 3.2 MIPS in the PEM. Thus, the number-crunching ratio suggests one might be able to tackle weather forecasting problems from 100-200 times the current problem and still get the answers to the users in the same amount of time. On the other hand, timeliness is a consideration. One might, in the interim, while waiting for satellite soundings, choose to calculate using moderate resolution and get the products disseminated in a more timely fashion. The results of these new efforts involving the implementation of the PEM on computers of the ILLIAC IV class will be reported on in a later paper.

REFERENCES

1 E MORENOFF W BECKETT P G KESEL F J WINNINGHOFF P M WOLFF
4-Way parallel processor partition of an atmospheric primitive-equation prediction model
Proceedings of the AFIPS SJCC 1971

2 P G KESEL F J WINNINGHOFF
Fleet numerical weather central's four-processor primitive equation model
Proceedings of the 6th AWS Technical Exchange Conference US Naval Academy Technical Report 242 1970 17–42

3 J SMAGORINSKY S MANAGE
L L HOLLOWAY JR
Numerical results from a 9-level general circulation model of the atmosphere
Monthly Weather Review Vol 93 No 12 1965 727–768

4 A ARAKAWA
Computational design for long term numerical integration of the equations of fluid motion: Two dimensional incompressible flow
Journal of Computer Physics Vol 1 1966 119–143

5 A ARAKAWA A KATAYAMA Y MINTZ
Numerical simulation of the general circulation of the atmosphere
Proceedings of the WMO/IUGG Symposium of NWP Tokyo 1968

6 W E LANGLOIS H C KWOK
Description of the Mintz-Arakawa numerical general circulation model
UCLA Dept of Meteorology Technical Report No 3 1969

7 N A PHILLIPS
A coordinate system having some special advantages for numerical forecasting
Journal of Meteorology Vol 14 1957

8 Y KURIHARA
Note on finite difference expression for the hydrostatic relation and pressure gradient force
Monthly Weather Review Vol 96 No 9 1968

9 E MORENOFF J B McLEAN
Job linkages and program strings
Rome Air Development Center Technical Report TR-66-71 1966

10 E MORENOFF J B McLEAN
Inter-program communications program string structures and buffer files
Proceedings of the AFIPS SJCC 1967 175–183

11 E MORENOFF
The table driven augmented programming environment: A general purpose user-oriented program for extending the capabilities of operating ststems
Rome Air Development Center Technical Report TR-69-108 1969

12 P G KESEL E MORENOFF
The Navy's operational four processor atmospheric prediction model
Proceedings of the NASA/ARPA ILLIAC IV Symposium Naval Postgraduate School Monterey California 1972

An analysis of optimal control system algorithms*

by CAROL N. WALTER

Xerox Corporation
Rochester, New York

and

GERALD H. COHEN

The University of Rochester
Rochester, New York

INTRODUCTION

Currently, there are methods available which were derived in the field of computer science to analyze and evaluate algorithms implemented in computer programs. The subject of this paper will involve a combination of three of these methods[1-3] with a rather rigorous simulation of three invariant imbedding algorithms in a manner strictly slanted toward their usefulness and importance in control system applications. The algorithms used to solve the problems and special solution formulations of the problems are presented first. Then, the numerical routines which provided the most efficient implementation of the problems in their algorithmic form are explained. And last, the adaptation of the analysis techniques to the problems is shown to aid in understanding the final conclusions drawn.

Some of the reasoning used in the selection of problems and the method of comparing the algorithms may or may not be totally applicable to algorithm analysis in other fields.

THE ALGORITHMS

The invariant imbedding algorithms used for this evaluation were derived[4-6] from the fundamental matrix specifically to provide numerical solutions for linear two-point boundary value problems. The principle of invariant imbedding was applied in the form of certain invariant matrices for solving subproblems imbedded in $x \in [x_0, L]$. The axis nomenclature used for expressing the operations in space in the imbedded area is described by Figure 1 ($\bar{x}_0$ indicates a variable left boundary of increasing thickness). The final computation in all three algorithms is the solution of the following transformed formulation of the state equation in terms of the two given boundary conditions:

$$U(x) = \psi_{11}(L, x, x_0) U(x_0) + J_1(L, x, x_0) \tag{1}$$

$$V(x) = \psi_{21}(L, x, x_0) U(x_0) + J_2(L, x, x_0) \tag{2}$$

Algorithm I (One-Sweep Transformation) integrates the following transmission, reflection and internal source differential matrix equations (3, 6; 4, 5; 7, 8), respectively, to provide values for the transformed matrix equations (1) and (2).

$$\dot{P}_{11}(x, x_0) = A_{11}(x) P_{11}(x, x_0) - P_{12}(x, x_0) A_{21}(x) P_{11}(x, x_0) \tag{3}$$

$$\dot{P}_{12}(x, x_0) = A_{11}(x) P_{12}(x, x_0) + A_{12}(x) - P_{12}(x, x_0) \cdot A_{21}(x) P_{12}(x, x_0) - P_{12}(x, x_0) A_{22}(x) \tag{4}$$

$$\dot{P}_{21}(x, x_0) = -P_{22}(x, x_0) A_{21}(x) P_{11}(x, x_0) \tag{5}$$

$$\dot{P}_{22}(x, x_0) = -P_{22}(x, x_0) A_{21}(x) P_{12}(x, x_0) - P_{22}(x, x_0) A_{22}(x) \tag{6}$$

$$\dot{\tilde{H}}_1(x, x_0) = F_1(x) + [A_{11}(x) - P_{12}(x, x_0) A_{21}(x)] \cdot \tilde{H}_1(x, x_0) - P_{12}(x, x_0) F_2(x) \tag{7}$$

$$\dot{\tilde{H}}_2(x, x_0) = -P_{22}(x, x_0) [F_2(x) + A_{21}(x)_1\tilde{H}(x, x_0)] \tag{8}$$

$$\text{Initial Conditions:} \quad P(x_0, x_0) = I, \ H(x_0, x_0) = 0 \tag{9}$$

$$\psi_{11}(L, x, x_0) = P_{11}^{-1}(L, x) P_{11}(L, x_0) \tag{10}$$

$$\psi_{21}(L, x, x_0) = P_{22}^{-1}(x, x_0) \cdot [P_{21}(L, x_0) - P_{21}(x, x_0)] \tag{11}$$

$$J_1(L, x, x_0) = P_{11}^{-1}(L, x) [\tilde{H}_1(L, x_0) - \tilde{H}_1(L, x)] \tag{12}$$

$$J_2(L, x, x_0) = P_{22}^{-1}(x, x_0) [\tilde{H}_2(L, x_0) - \tilde{H}_2(x, x_0)] \tag{13}$$

* This research was supported in part by the Office of Naval Research under contract number N00014-68-0091. Such support does not imply endorsement of the content by the Department of Navy.

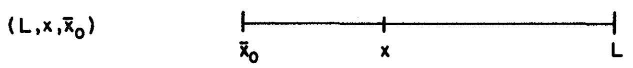


Figure 1—"Medium" nomenclature

The following four steps required for computing algorithm I are pictorially represented in Figure 2.

(1) Integrate equations (3-8) from x_0 to x with the initial conditions described by equation (9) applied at x_0, and store $P_{21}(x, x_0)$, $P_{22}(x, x_0)$ and $\tilde{H}_2(x, x_0)$ at each x.
(2) Adjoin equations (3), (4), (7) with initial conditions (equation 9) applied at x, and integrate from x to L to obtain $P_{11}(L, x)$, $P_{12}(L, x)$ and $\tilde{H}_1(L, x)$ $\forall x$.
(3) Integrate the complete set of equations from x_0 to L to obtain the necessary values for equations (10-13).
(4) Solve equations (1) and (2) $\forall x$.

Thus, the solution for each point is available after all of the sweeps (one sweep for each data point) have been completed.

Algorithms II (Two-Sweep Riccati) and III (One-Sweep Riccati) integrate the following Riccati differential equations in the process of their solution steps.

$$\begin{aligned}\dot{S}_{21}(x, L) = A_{21}(x) + A_{22}(x) S_{21}(x, L)& \\ - S_{21}(x, L) A_{11}(x)& \qquad (14)\\ - S_{21}(x, L) A_{12}(x) S_{21}(x, L)&\end{aligned}$$

Initial Conditions: $S_{21}(L, L) = 0$,

where: $\psi_{21}(L, x_0, \bar{x}_0)\ |_{\bar{x}_0 = x_0} = S_{21}(x_0, L)$.

$$\begin{aligned}\dot{H}_2(x, L) = F_2(x) - S_{21}(x, L) F_1(x)& \\ + A_{22}(x) H_2(x, L)& \qquad (15)\\ - S_{21}(x, L) A_{12}(x)\ H_2(x, L)&\end{aligned}$$

Initial Conditions: $H_2(L, L) = 0$,

where: $J_2(L, x_0, \bar{x}_0)\ |_{\bar{x}_0 = x_0} = H_2(x_0, L)$.

Algorithm II implements the following steps: (Consult Figure 3 for the flow diagram.)

(1) The first sweep, a backward sweep from $L \rightarrow x_0$, requires that equations (14) and (15) be integrated backward in space to enable equation (2) to be solved for $V(x_0)$.
(2) The problem now becomes an initial value problem (see Eq. 2). Therefore, the second sweep is a forward integration of the solution differential equations: $\dot{U}(x)$ and $\dot{V}(x)$ from $x_0 \rightarrow L$.

The additional equations (16), (17), (18), (19) are necessary to compute algorithm III (One-Sweep Riccati).

$$\frac{\partial \psi_{11}(L, x, \bar{x}_0)}{\partial \bar{x}_0} = -\psi_{11}(L, x, \bar{x}_0)[A_{11}(\bar{x}_0) + A_{12}(\bar{x}_0)\psi_{21}(L, \bar{x}_0, \bar{x}_0)]\,|_{x=\bar{x}_0} \qquad (16)$$

Initial Conditions: $\psi_{11}(L, \bar{x}_0, \bar{x}_0) = I$.

$$\frac{\partial \psi_{21}(L, x, \bar{x}_0)}{\partial \bar{x}_0} = -\psi_{21}(L, x, \bar{x}_0)[A_{11}(\bar{x}_0) + A_{12}(\bar{x}_0)\psi_{21}(L, \bar{x}_0, \bar{x}_0)]\,|_{x=\bar{x}_0} \qquad (17)$$

Initial Conditions: $\psi_{21}(L, \bar{x}_0, \bar{x}_0) = S_{21}(\bar{x}_0, L)$

$$\frac{\partial J_1(L, x, \bar{x}_0)}{\partial \bar{x}_0} = -\psi_{11}(L, x, \bar{x}_0)[F_1(\bar{x}_0) + A_{12}(\bar{x}_0) J_2(L, \bar{x}_0, \bar{x}_0)]\,|_{x=\bar{x}_0} \qquad (18)$$

Initial Conditions: $J_1(L, \bar{x}_0, \bar{x}_0) = 0$.

$$\frac{\partial J_2(L, x, \bar{x}_0)}{\partial \bar{x}_0} = -\psi_{21}(L, x, \bar{x}_0)[F_1(\bar{x}_0) + A_{12}(\bar{x}_0) J_2(L, \bar{x}_0, \bar{x}_0)]\,|_{x=\bar{x}_0} \qquad (19)$$

Initial Conditions: $J_2(L, \bar{x}_0, \bar{x}_0) = H_2(\bar{x}_0, L)$

Equations (16)-(19) hold $\forall \bar{x}_0 \leq x$.

Therefore, the steps required (consult Figure 4 for

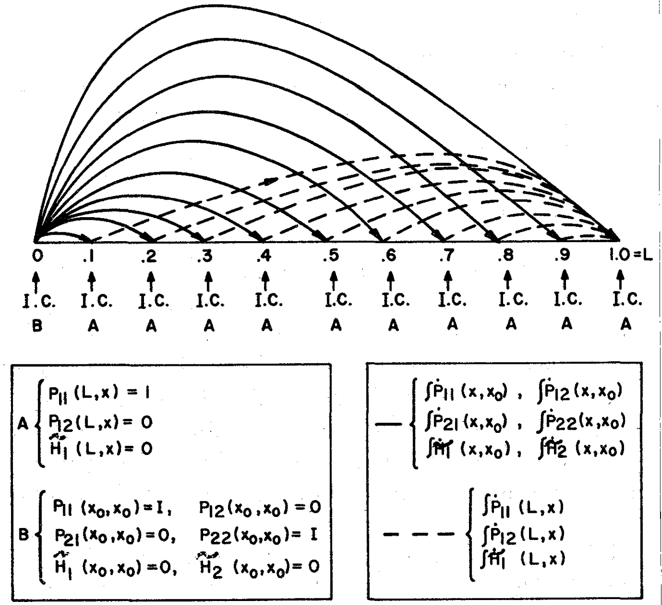

0
.1
.2
.3
.4
.5
.6
.7
.8
.9
1.0=L
I.C.
I.C.
I.C.
I.C.
I.C.
I.C.
I.C.
I.C.
I.C.
I.C.
I.C.
B
A
A
A
A
A
A
A
A
A
A
A { P11 (L,x) = I, P12(L,x)= 0, H̃1 (L,x)= 0
B { P11 (x0,x0)=I, P12(x0,x0) = 0, P21(x0,x0)=0, P22(x0,x0)= I, H̃1 (x0,x0) =0, H̃2 (x0,x0)= 0
— { ∫Ṗ11 (x,x0), ∫Ṗ12(x,x0), ∫Ṗ21 (x,x0), ∫Ṗ22(x,x0), ∫H̃̇1 (x,x0), ∫H̃̇2 (x,x0)
- - - { ∫Ṗ11 (L,x), ∫Ṗ12(L,x), ∫H̃̇1 (L,x)

Figure 2—Algorithm I-one-sweep transformation

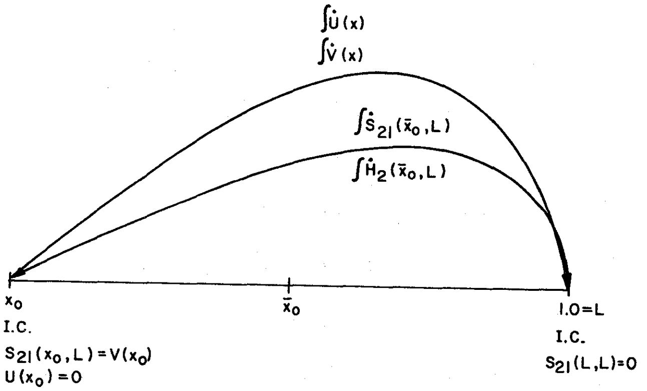


Figure 3—Algorithm II-two-sweep Riccati algorithm

the flow diagram) to compute solutions to algorithm III are:

(1) Integrate the Riccati equations (14) and (15) from L backwards to x. At x adjoin equations (16) and (17) (where $\psi_{21}(L, x, \bar{x}_0 = S_{21}(x, L))$ and equations (18) and (19) (where $J_2(L, x, \bar{x}_0) = H_2(x, L)$). Integrate all six equations backward from x to x_0.

(2) Equations (1) and (2) produce an immediate solution for each x sweep for $U(x)$ and $V(x)$.

(3) Continue until the solution $\forall x \in [x_0, L]$ has been obtained.

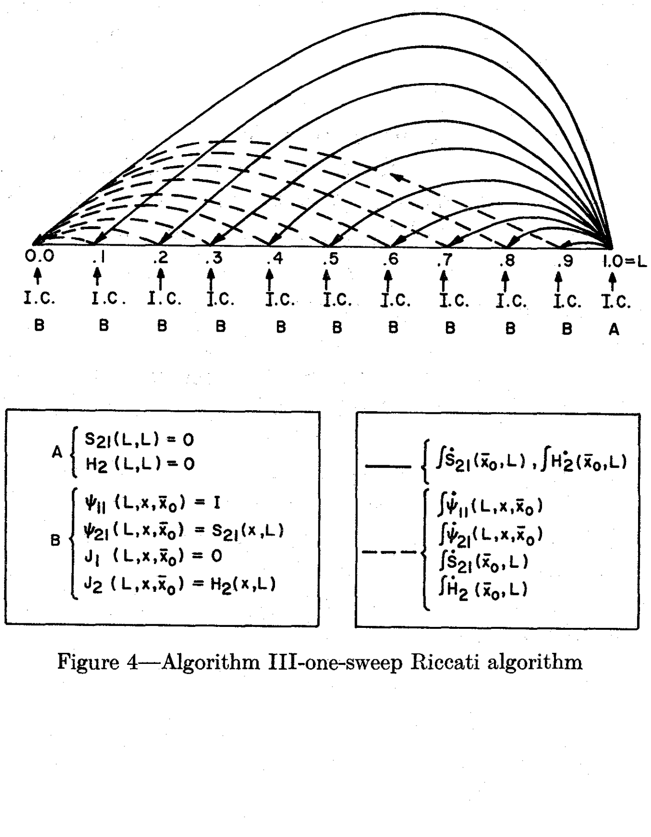

0.0
.1
.2
.3
.4
.5
.6
.7
.8
.9
1.0=L
I.C.
B
A
$S_{21}(L,L)=0$
$H_2(L,L)=0$
$\psi_{11}(L,x,\bar{x}_0)=I$
$\psi_{21}(L,x,\bar{x}_0)=S_{21}(x,L)$
$J_1(L,x,\bar{x}_0)=0$
$J_2(L,x,\bar{x}_0)=H_2(x,L)$
$\int \dot{S}_{21}(\bar{x}_0,L), \int \dot{H}_2(\bar{x}_0,L)$
$\int \dot{\psi}_{11}(L,x,\bar{x}_0)$
$\int \dot{\psi}_{21}(L,x,\bar{x}_0)$
$\int \dot{S}_{21}(\bar{x}_0,L)$
$\int \dot{H}_2(\bar{x}_0,L)$

Figure 4—Algorithm III-one-sweep Riccati algorithm

PROBLEM IMPLEMENTATION CONSIDERATIONS

Two problems were chosen to provide a worst case and a best case digital simulation of each algorithm. Problem I is the reduced system of ordinary differential equations for a lumped parameter control problem.

$$\begin{bmatrix}\dot{u}(x)\\ \dot{v}(x)\end{bmatrix} = \begin{bmatrix}-1 & -1\\ -2 & 1\end{bmatrix}\begin{bmatrix}u(x)\\ v(x)\end{bmatrix}$$

Initial Conditions: $u(x_0) = 1$

$$v(L) = 0$$

This problem does not require the matrix formulation property or the forcing function of the algorithms. Therefore, the computational form of each algorithm for Problem I will be of minimal complexity. Problem II, the worst case application, is a distributed optimal control system in the form of a hyperbolic system of partial differential equations.

$$\frac{\partial u(x,t)}{\partial t} + \frac{\partial u(x,t)}{\partial x} = -u-v \qquad (20)$$

$$\text{I.C.}\ \left\{\begin{array}{ll} u(x,0) = h(x) = 1, & x \in [0, L] \\ u(0,t) = g(t) = 1, & t \in [0, t] \end{array}\right\}$$

$$\frac{\partial v}{\partial t} + \frac{\partial v}{\partial x} = -u+v \qquad (21)$$

$$\text{I.C.}\ \left\{\begin{array}{ll} v(x,T) = 0, & x \in [0, L] \\ v(L,t) = 0, & t \in [0, T] \end{array}\right\}$$

The method of lines is used to transform the partial differential equations into a set of ordinary differential difference equations. Equations (20) and (21) are discretized in the time variable by the unique substitution of the forward difference approximation for $\partial u(x, t)/\partial t$ and a backward difference for $\partial v(x, t)/\partial t$. Therefore, the resulting solution equations for algorithmic computation are inhomogeneous and also require the matrix formulation of the algorithms (maximal algorithm implementation).

Forward Difference $$\frac{du_{i+1}(x)}{dx} = \frac{-1}{\Delta t}[u_{i+1}(x) + u_i(x)] - u_{i+1}(x) - v_{i+1}(x) \qquad (22)$$

Backward Difference $$\frac{dv_i(x)}{dx} = \frac{-1}{\Delta t}[v_{i+1}(x) + v_i(x)] - u_i(x) + v_i(x) \qquad (23)$$

where $i = 0, \ldots, N-1$, N = number of intervals between x_0 and L.

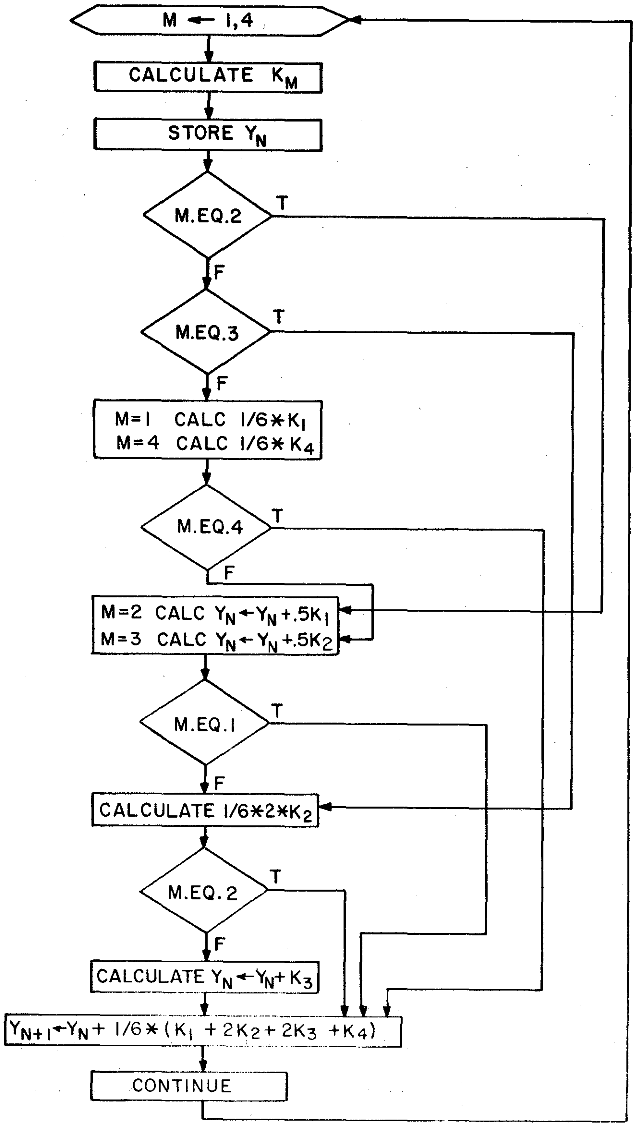


Figure 5—Runge-Kutta routine flow chart

Problems I and II can be solved for the same data points along the x axis by realizing that when the method of characteristics is used for Problem II, it becomes identical to Problem I. All integration in Problem I is along the x axis; in Problem II these data points exist on the characteristic diagonal line (length$=1$).

NUMERICAL ROUTINES

The development of the actual computational form of the algorithms for both problems required a rather efficient manner of performing a fourth-order Runge-Kutta integration which could be adapted to the matrix formulations necessary in Problem II. The classical fourth-order Runge-Kutta technique[7] implemented follows:

$$y_{n+1}=y_n+\tfrac{1}{6}(k_1+2k_2+2k_3+k_4);$$

$$k_1=hf(x_n,y_n)$$

$$k_2=hf(x_n+\tfrac{1}{2}h,\ y_n+\tfrac{1}{2}k_1)$$

$$k_3=hf(x_n+\tfrac{1}{2}h,\ y_n+\tfrac{1}{2}k_2)$$

$$k_4=hf(x_n+h,\ y_n+k_3).$$

A special in-line Runge-Kutta routine (outline) was developed (refer to Figure 5 for a flow chart of this routine) to avoid the overhead of calling a subroutine and to make optimum use of the following two facts inherent in the integratable equations:

(1) The independent variable x_n never appears to the right of the equals sign.
(2) The coupled property of the invariant imbedding algorithm equations produces functions which are constant within an integration interval of the dependent variable. These functions change at a fixed value of the independent variable as a function of the interval on which the boundary value problem is specified.

A matrix inversion routine was required to implement algorithm I in Problem II. The IBM Scientific Subroutine MINV, which performs a matrix inversion by the Gauss Jordon Method with a Full Maximal Pivoting technique, was chosen for this requirement due to the following facts:

(1) An accurate matrix inversion technique is more complex than an ordinary in-line routine.
(2) The inversion was not required extensively throughout the program for algorithm I.

ANALYSIS TECHNIQUES**

The three methods of analysis implemented in the problem's computing characteristics were:

(1) solvability analysis;
(2) local time and storage analysis;
(3) efficiency and optimality analysis.

** All computations referred to in this section were made on an IBM 360/65 computer in Fortran G.

TABLE I—Digital Analysis—Problem I

Algorithms Problem I	Number of Executable Statements	Execution time (sec)	Execution Program Space (Bytes)	Storage Program Space + Work Space (Bytes)	Maximum Absolute Error ME = $\lvert E\rvert_{max}$	Δx
I	106	0.09	22,288	26K	0.0000092	0.1
II	68	0.04	21,440	26K	0.0000476	0.1
III	73	1.21	21,600	26K	0.0004870	0.00625

Solvability Analysis

The first technique provides the basis for all of the analysis performed. The accuracy of the result of each algorithm is compared to the analytic solution of the respective problem, to determine if the algorithm yields the correct solution. The numerical method of error computation is used to obtain the error for each point in the solution.

ERROR = E = Analytic solution value—algorithm value.[1] The second indication of the computational workability of a problem is its stability. Algorithms I and III transform an unstable set of solution (system) equations to a stable set for their necessary integrations. However, algorithm II uses the original problem equations to obtain the final solution once the initial conditions have been computed. These equations may be unstable.

Local time and storage analysis

A local analysis[2] to compare algorithms is one which investigates the important characteristics of some algorithm under "worst case" and "best case" input conditions. Therefore, using a local analysis, the three algorithms presented are evaluated in terms of execution time and storage for both problems. The criteria for determining the point of comparison for the three algorithms for each problem was chosen as the value of Δx (Runge-Kutta integration increment) where they exhibit the same chosen maximum absolute error (ME) (decimal accuracy) for a certain number of data points.***

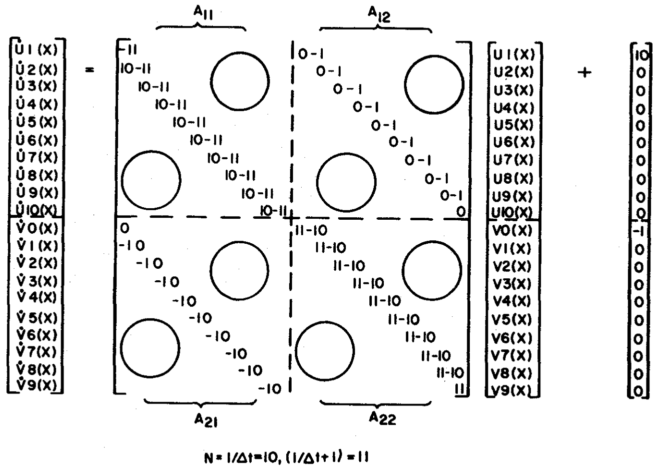
$$ME = \lvert E \rvert_{max}$$

A three-place-decimal-accuracy, local analysis for Problem I is shown in Table I.

In Problem II the same approximate ($\lvert E \rvert_{max} = 0.074$) one-decimal place of accuracy comparison with the analytic solution was the criteria for comparison because for $\Delta t < 0.1$ (discrete time increment), the matrices became ill conditioned in Algorithm I. Refer to Figure 6 for the matrix form of the solution equations for Problem II. Conditioning the matrices would only cause further computational complexity and longer execution time (cost) and add nothing to the comparison besides increased decimal accuracy (refer to Table II for the local analysis results).

Efficiency and optimality

The efficiency-optimality analysis is presented in Pager[3] in the theoretical terms of a Turing Machine. The theory begins with the basic premise that a function $f^{(n)}$ is partial recursive over a set of arguments S. This


A11
A12
A21
A22
U̇1(X) U̇2(X) U̇3(X) U̇4(X) U̇5(X) U̇6(X) U̇7(X) U̇8(X) U̇9(X) U̇10(X)
V̇0(X) V̇1(X) V̇2(X) V̇3(X) V̇4(X) V̇5(X) V̇6(X) V̇7(X) V̇8(X) V̇9(X)
=
-11
10-11
0-1
0
11-10
-10
11
U1(X) U2(X) U3(X) U4(X) U5(X) U6(X) U7(X) U8(X) U9(X) U10(X)
V0(X) V1(X) V2(X) V3(X) V4(X) V5(X) V6(X) V7(X) V8(X) V9(X)
+
10 0 0 0 0 0 0 0 0 0
-1 0 0 0 0 0 0 0 0 0

N = 1/Δt = 10, (1/Δt + 1) = 11

Figure 6—Matrix formulation

*** kth decimal accuracy (significant places) with the analytic solution[1] is defined as: $\lvert E\rvert \leq 1/2 \times 10^{-k}$.

TABLE II—Digital Analysis

ALGORITHMS	ΔX	NO. OF EXECUTION TABLE STATEMENTS	EXECUTION TIME(1) (SEC.)	EXECUTION PROGRAM SPACE (BYTES)	STORAGE PROGRAM SPACE + WORK SPACE(2) (BYTES)	MAXIMUM ABSOLUTE ERROR ME = $\lvert E\rvert_{max}$	SPACE · TIME MEASURE (BYTE · SEC.) $\mu(Z_i,x_1,\ldots x_n)$ i = # of ALGORITHM
PROBLEM I							
TWO-SWEEP RICCATI	0.1	68	0.04	21,440	26K	0.0000476	1,040
ONE-SWEEP RICCATI	0.1	73	0.11	21,528	26K	0.0073400	2,860
ONE-SWEEP TRANSFORMATION	0.1	106	0.09	22,288	26K	0.0000092	2,340
PROBLEM II							
TWO-SWEEP RICCATI	0.05	227	8.35	37,816	56K	0.0744596	467,600
ONE-SWEEP RICCATI	0.025	300	294.04	53,336	60K	0.0740608	17,642,400
ONE-SWEEP TRANSFORMATION	0.05	351M 91S	169.24	95,904	100K	0.0746680	16,924,000

(1)ACCURATE TO ± 0.01 SECONDS (via University of Rochester Computing Center, Subroutine TIMER)

(2) ACCURATE TO ± 2 K BYTES

means that there is a Turing Machine $Z \ni$

$$f^{(n)}(x_1, \ldots, x_n) = U[\min_y T_n(z, x_1, \ldots, x_n, y)]$$
$$\forall (x_1, \ldots, x_n) \in S, \qquad (24)$$

where Z calculates f over S. Consulting Davis,[8] this infers that not all input arguments will allow the computation in a certain Turing Machine Z to go to completion. Therefore, only certain arguments, or certain problems, provide input strings to Z which can be successfully computed. The timing involved to compute argument $(x_1, \ldots, x_n)$ for Turing Machine Z is $E(z, x_1, \ldots, x_n)$. The space $M(z, x_1, \ldots, x_n)$ is the sum of the problem space and the work space. The probability figure $p(x_1, \ldots, x_n)$ is the probability that it will be necessary to compute $f(x_1, \ldots, x_n)$. Then, $p(x_1, \ldots, x_n)$ is >0 only for $(x_1, \ldots, x_n)$ in the domain of f. The space-time measure $\gamma_p(z)$ of a Turing Machine Z is given by the following relation:

$$\gamma_p(z) = \frac{1}{c}\left[\sum_{x_1,\ldots,x_n=0}^{\infty} p(x_1, \ldots, x_n)\mu(z, x_1, \ldots, x_n)\right] \qquad (25)$$

where c = number of computations (problems) (arguments) that satisfy equation (24).

The function $\mu(z, x_1, \ldots, x_n)$, the space-time measure of a computation of a Turing Machine Z for argument (problem) $(x_1, \ldots, x_n)$, is an increasing recursive function of both $E(z, x_1, \ldots, x_n)$ and $M(z, x_1, \ldots, x_n)$.

Application of efficiency-optimality

For an algorithm (computer program), a practical application of this theory is used. Since there are certain input arguments for which the computation will not go to completion, only two problems (arguments) are used for each of the three algorithms (computations) to obtain the space-time measure $\gamma_{p_{\zeta_c}}(z_i)$. In this practical usage, the following application of the theory is implemented:

(1) z_i represents each algorithm; $i=1, 2, 3$.

(2) ζ_j represents the problems (arguments) that are included in the partially recursive alphabet; $j=1, c=2$.

(3) $p_{\zeta_j}(x_1, \ldots, x_n)$ for each ζ_j are assumed to be ≈ 1 and equal to each other, $(p_{\zeta_1}(x_1, \ldots, x_n) \approx p_{\zeta_2}(x_1, \ldots, x_n)$. ζ_1 represents Problem I, the simplified form of an extensively used control problem, a "best case" application of the algorithms. Problem II (ζ_2) represents the complex version of an optimal control problem used quite extensively in the field of chemical engineering, a "worst case" application of the algorithms. This approximation has been made since it would be very time consuming to obtain a statistical calculation of the probability of usage for these problems. Therefore, the effect of the probability in the calculation of $\gamma_{p_{\zeta_c}}(z_i)$ is represented by a constant k_j.

(4) The ζ_j for each algorithm were written in a very efficient manner (separate programs of the same algorithm for each problem). Therefore, the program space changes for each ζ_j to more efficiently implement the algorithms. This procedure agrees with Pager's[3] definition of two Turing machines having the same behavior if they perform the same sequence of tasks for each argument.

Equation (27) is now expressed in the following generalized form:

$$\text{Optimality of } z_i=\gamma_{p_{\zeta_c}}(z_i) = \frac{1}{c}\left[\sum_{j=1}^{c=2} k_j\mu(z_i, x_1, \ldots, x_n)\right]. \quad (26)$$

$$\text{Efficiency of } z_i = \frac{1}{\gamma_{p_{\zeta_c}}(z_i)}. \quad (27)$$

Then, a local comparison of the efficiency-optimality of the three algorithms is performed using the maximum error criteria (ME) (approximately one-decimal-place accuracy). Refer to Tables II and III for the efficiency-optimality results for both problems.

TABLE III—Efficiency-Optimality Analysis

ALGORITHMS	OPTIMALITY (SPACE TIME MEASURE) $\gamma_{p_{\zeta_c}}(z_i)$	EFFICIENCY $1/\gamma_{p_{\zeta_c}}(z_i)$
ONE-SWEEP TRANSFORMATION	8,498,170	0.01176×10^{-5}
TWO-SWEEP RICCATI	234,320	0.4267×10^{-5}
ONE-SWEEP RICCATI	8,822,630	0.01132×10^{-5}

RESULTS

It is apparent that the problems chosen are solvable with these algorithms since the solutions are stable within the boundary conditions chosen, even though both problems have unstable characteristic roots. The local storage and time analysis proved that algorithm II required the minimum execution storage and time for both problems, due to its minimal amount of manipulation of original data; algorithm III required the maximum storage and algorithm I, the maximum time. Algorithm I was the most accurate when a comparison was made for a certain decimal accuracy in Problem I. The efficiency-optimality analysis indicates that algorithm II is the most optimum and efficient (smallest optimality measure and largest efficiency measure). Algorithms I and III, respectively, are second and third in optimality and efficiency. A slight anomaly results here because algorithm II should be less accurate than algorithms I or III for a given Δx (Runge-Kutta integration increment) since the integration performed is with unstable solution equations. This is true for algorithm I in Problem I and for algorithm III in Problem II. Evidently, the three types of error encountered in these digital solutions, original data error, roundoff error, and truncation error, mask the theoretically proven instability of algorithm II in comparison with algorithms III and I. This is logically deduced since the manipulations of algorithms III and I are subject to the largest amount of original data-error buildup of the three algorithms. The error buildup for algorithm II would have been substantially larger if the imbedding interval was greater than $x=1$.

Finally, it can be stated that the two types of storage-time analysis yielded consistent results and, therefore, either method would have sufficed.

REFERENCES

1 A RALSTON
A first course in numerical analysis
(New York) McGraw-Hill Book Company 1965

2 D E KNUTH
Mathematical analysis of algorithms
Stanford University Computer Science Department
STAN-CS-71-206 March 1971

3 D PAGER
On the efficiency of algorithms
Journal of the ACM Vol 17 No 4 October 1970 pp 708-714

4 E D DENMAN G H COHEN
One and two sweep methods of solving linear two-point boundary value problems
USC Department of Electrical Engineering
Technical Report No 70-39 August 1970

5 G H COHEN C N WALTER
Hybrid computer solutions of partial differential equations using invariant imbedding techniques
Sixth Annual Princeton Conference on Information Sciences and Systems March 1972

6 C N WALTER
An analysis of two-point boundary value problem algorithms
University of Rochester Department of Electrical Engineering Master's Essay December 1971

7 C F GERALD
Applied numerical analysis
(Philippines) Addison-Wesley Publishing Company 1970

8 M DAVIS
Computability & unsolvability
(New York) McGraw-Hill Book Company 1958

Computer simulations of the metropolis

by BRITTON HARRIS

University of Pennsylvania
Philadelphia, Pennsylvania

The history of modern computer simulation of urban affairs represents the confluence of a number of trends which came to maturity in the middle of this century. Probably the oldest of these tendencies is the emphasis on planned urban development which has existed for millennia and which in the last century has demonstrated considerable vitality as a reaction to the excesses of the industrial revolution and the poverty and squalor of nineteenth-century cities. A second strand is the development of economic and sociological theory which goes a considerable distance in explaining some aspects of the organization and form of metropolitan settlement and its growth. These theories have a long history, but have matured principally during the 1920's and 1930's. Finally, as a methodological catalyst, the development of the automobile, of a Federal Bureau of Public Roads dedicated to providing facilities for it, and of the large-scale metropolitan study based on the origin-and-destination survey have together made possible the crystallization and further growth of simulation methods. These methods are thus proximately based on the engineering attitude and computer technology of the large-scale transportation study, but they are in a position to draw on a number of other important streams of intellectual development.

The transportation planning effort as carried out in large metropolitan area studies produced or laid the basis for three major advances in planning methods, all related to simulation. First, through the use of origin-and-destination studies and through the consideration of small-area detail, these studies emphasized the creation of large data banks. The bringing to perfection of such data banks has become a matter of nagging concern in the fields of urban management and urban planning, but for a variety of reasons adequate reservoirs of data have not yet been accumulated. Data is incommensurate as to area definition and activity definition. It is uneven in coverage across political jurisdictions. It lacks important elements such as detailed employment location and accurate descriptions of man-made structures. There are no time series data and diverse data sets are frequently available for years which do not match. The major transportation studies have solved most of these problems (except time series data) on a one-shot basis. Second although it is frequently not recognized, transportation studies have taken an essentially behavioral view of transportation demand, although a very naive one. Over the last ten to fifteen years, there has been a growing recognition that a behavioral understanding of the reasons why certain decisions (to travel, to move, to build, etc.) are made and how they are influenced by the environment and by public policy is the key to a useful understanding of the urban organism. Such understandings are being expanded from the simple descriptive level to more subtle and complex views of more diverse and extensive types of behavior. Third, on the basis of these data and a limited behavioral understanding, transportation analysts were able to construct very large-scale simulation models of transportation behavior. These models can predict the use of transportation systems in substantial detail under varying assumptions. In considering the merits of this achievement, one must note the very large size of the systems involved and the fact that these systems have been treated in a fairly holistic fashion. Two special aspects of this whole development deserve slightly more extended discussion.

We should have expected, since we are discussing transportation planning, that the developments of the 50's and 60's could have produced a very extensive improvement in plan-making methods themselves. The simulation models which I have referred to under the third point above are essentially models for predicting behavior and the impacts of change and policy on these predictions. Almost nothing in the transportation literature bears on the question of producing a plan. One might have expected that the engineering approach to transportation planning would have generated optimizing techniques based either on the analytical solution of the conditions for an extremum or on search methods defined in some form of mathematical programming. Actually examples of this are extremely rare, and transportation planning takes the form of the evaluation of a limited number of alternatives which are generated in very conventional ways.

The other observation is almost superfluous, having to do with the utilization of computers. The very large masses of data which are available for any city, and particularly as the outcome of a transportation study, moved research rapidly from punched card storage to tape storage and computer manipulation. At the same time, with increasing computer power, the analyses which were conducted became more sophisticated. Finally, the very large simulation models themselves require, for a transportation system alone, computer time on the order of hours rather than tenths of hours. Inevitably the appetite of simulation model designers requires more and more core and frequently more and more computer time.

Against this background, let me discuss briefly several different dimensions of variation which apply to the computer simulation of urban growth and change.

I assume that properly designed computer simulations can be used in a two-edged way—either as an aid to scientific investigation or as a means of making predictions which will vary under different assumptions about the state of the real world, the growth of technology, and the policies which are pursued by government, corporations, and households. I take the general view that policy manipulations are becoming increasingly disaggregated both as to the means which they employ and the objectives which they pursue. This means essentially that the most useful sets of simulation methods may have to do with a fairly detailed portrayal of the phenomena. I believe that there is room in general for considerable skepticism as to the accuracy of the simulation models which are used for policy explorations. In the extreme case, one may fall back on the alternative view that, even with inaccurate predictions, the use of models helps to define the nature of the problem and the construction of models helps to develop deeper insights into the theoretical and practical issues which are involved.

The essential advantages of large-scale computer simulation models lie, first of all, in their extensive bookkeeping and computational capabilities. These aspects may escape direct theoretical comprehension and hand manipulations. By extension, computer-based models can in principle take into account extensive interactions between different parts of the urban system and can trace the repercussions of events widely over space and time. This capability clearly depends on the ability of the analyst to identify the interactions in the first instance.

We may now turn to two principal aspects of the subject matter which are dealt with in these computer simulations.

The first distinction has to do with the difference between inter-urban and intra-urban simulations. Inter-urban and inter-regional simulations are necessary to provide a basis for action in any particular sub-area of a large country. Projections of the probable growth of the Philadelphia metropolitan region or of the State of California, hopefully under various policies, is a necessary background to planning for the metropolis or the state. Such projections are best made in the multiregional context so as to take into account the competition and interactions which occur at the national scale. Single projections, including those proposed by J. H. Forrester in *Urban Dynamics*, are extremely unreliable because they isolate the entity from its environment. Projections in this class fall into the realm of regional geography, regional science, and classical locational economics. I personally am much more concerned with intra-metropolitan locational patterns, given the prior determination of levels of growth, composition of industry, and income. It is of course true that certain internal decisions affect these levels of growth, but in my view there is not yet an adequate basis for modeling this feedback. I am concerned therefore in the balance of this paper principally with the interaction between intra-metropolitan policies of all types and the growth and development of the metropolis within its own generously defined boundaries.

A second major distinction can be made along a spectrum of phenomena—physical, economic, social and political. As we move along this continuum, phenomena become more and more difficult to simulate because the theoretical models which describe them become less and less quantitative and to an extent more purely descriptive. In the physical realm, for example, and including the physical development of the biosphere, we can simulate fairly well such matters as hydrology, meteorology, and pollution. We can also, as I have suggested, simulate economic behavior such as the use of transportation facilities and choices of residential location. There are difficulties in these predictions which arise not from our lack of understanding of the phenomena, but from the existence of externalities which make certain aspects of projections more dependent on large-scale random events. Many of the behaviors of businesses and households in the metropolitan area are at least quasi-economic; their use of public facilities, for example, can be interpreted in the paradigm of economic behavior. Nevertheless, as activities become increasingly social, as in the achievement and employment of education, skills, and upward mobility, prediction becomes more difficult and less accurate. Similar and stronger remarks can be made about political behavior. Finally, social, political, and racial considerations interact with many otherwise straightforward phenomena. The rise and fall of neighborhoods, the preservation and deterioration of the housing stock, and many other economic or quasi-economic behaviors are in the city immersed in these higher-level social systems.

It should be apparent that a thread constantly running through all of these subject matters is the location of activities in space, their competition for sites, and their interaction with other activities both near and far by transportation and communication. Our theories and models of communication are much weaker than our theories and models of transportation. It is difficult, however, to see how we can possibly separate the overwhelming majority of the urban phenomena that we wish to simulate from their spatial distribution or from their interactions. Communication and transportation models therefore occupy a central place in the simulation process.

Simulation in the sense in which I discuss it here does not consist at all of Monte Carlo or single-event simulations and, indeed, has very few stochastic properties. Some, and indeed a majority, of existing models assume some sort of probabilistic laws governing human behavior, but such large numbers of individuals are being dealt with the division of people amongst various modes of behavior is in itself deterministic. In consequence of this type of assumption, the outcome of two successive runs of most of these models with the same inputs would be identical. I personally believe that this is desirable because large-scale events which might drastically alter the evolution of a metropolitan area should properly be explicitly under the control of the investigator. It follows from the foregoing discussion that many simulation models could be expressed in analytical form. Owing, however, to their very large size and nonlinearity, the solution of the analytical form of the models is usually outrageously difficult. A great deal of the computation involved in simulation is therefore one or another form of iterative solution to large and complex systems.

The probabilistic interpretation of behavior is intimately related to questions of disaggregation and reaggregation, and to the distinction between descriptive and behavioral models. A simple example would be the analysis of the distribution of shopping trips amongst shopping centers. A linear programming solution would assign most individuals to the nearest center, but this is obviously not what takes place. The original models dealing with phenomena of this type were descriptive in the sense that they attempted to replicate behavior without paying detailed attention to people's motives and decision-making processes. These matters are now coming under increased scrutiny, with the result that the analyst is faced by a bewildering array of behaviors and attitudes. Quite apparently, while this understanding of behavior may provide in principle a sounder basis for the construction of simulation models, it must be accompanied by the evolution of rules of aggregation which govern the deduction of mass phenomena from individual behavior. In principle, any model, no matter how highly aggregated, should have been derived in any one of a number of possible ways from an understanding of the behavior of decision-making units. This is a profound and complex problem which has only begun to receive adequate attention.

Probably the most interesting, difficult, and subtle of all of the issues involved in modeling revolves around the question of static versus dynamic models. This problem affects the basic structure of models, the mode of simulation, and the types of policy conclusions which can be drawn. Directly and indirectly the issues involved appear in many of the disputes which arise from modeling.

The more sophisticated computational, econometric, and mathematical discussions which arise over this issue have a somewhat more naive but very useful counterpart in the planning profession. Quite simply, twenty years ago the profession was strongly oriented toward the production of a "comprehensive plan" which envisaged some future state of affairs toward which the efforts of planned development should be directed. This conception has been roundly criticized on many grounds. The definition of a future state apparently left no room for further development beyond that date. The preoccupation with future conditions left present difficulties untended. The future state might indeed be incapable of achievement either because it was too costly, or because institutional obstacles existed, or because the path to it might be blocked by the behavior of individual decision-makers. In the light of all of these and many other criticisms, the idea of the comprehensive plan has fallen into some disrepute, and much more attention has been thrown on planning methods which emphasized the path of development, the most immediate measures which will relieve present difficulties, and the modes by which various segments of the population are impacted by and involved in the planning process and the implementation of plans. This second procedure is much more process-oriented and more apparently socially aware. It is also more oriented to the practical problems of consensus and implementation, and hence apparently more realistic.

In my own view, the planning orientations of these two approaches are complementary rather than competitive. The difficulty with the more recent and more dynamic view is that it is not guaranteed to lead to an acceptable or viable future state. It does not provide any means of defining an image of the future toward which the metropolis and its population can aspire. As an experimental vehicle, a dynamic model could be very clumsy since it is "forward-seeking" rather than "backward-seeking." On the other hand, the comprehensive plan defines some future optimal state subject to the possible difficulty that "you can't get there from here."

The relationship of these two approaches to formal and mathematical aspects of modeling should be apparent. The comprehensive planning model is entirely compatible with the idea of optimizing through a mathematical program. It turns out that the locational and organizational problems of cities are incredibly difficult to solve in this mode because they provide very large-scale, non-linear, integer programming problems with many local optima. Nevertheless, viewing the problems in this light provides important insights into design methodology. In certain cases, portions of the problem may be quite properly cast in a programming format. This is especially true of the predictive portion of models where market behavior is involved. Generally, however, even here the models are best solved by iterative procedures.

The dynamic approach to planning corresponds in an intuitively simple way with dynamic or growth models cast in the form of differential or difference equations. Once again, the typical system of equations would be extremely large, non-linear, and complicated. Models of this type are supposedly represented most clearly by the Dynamo system of J. H. Forrester, but in fact versions of such models have been used in many types of forecasting for metropolitan areas long prior to Forrester's *Urban Dynamics.* Such models have a verisimilitude which makes them very popular with professional planners and decision-makers, and an element of mathematical sophistication which makes them attractive to operations researchers and analysts. As I have sketched above, their operation is somewhat difficult for purposes of policy testing. At the same time, their calibration is particularly difficult from the point of view of data requirements because at least two points in time are required in fine-area detail. From an econometric point of view, many more data points would be desirable, but this is in general utterly impractical.

There are various points of contact between static optimizing models and dynamic models. One of these is entirely utopian in the present state of the art, but should be borne in mind as a future possible line of development. This would be to use the criterion function of an optimizing model to optimize not over metropolitan arrangements (as in the static case), but over the choice of policies, using the dynamic model as an embedded predictor in the dynamic programming context. Because of the very large number of possible policy combinations, this approach is presently infeasible, but it may have some future value.

More practically, the relation between dynamic and static models may be explored along a different line. The performance of a dynamic model may be regarded as an effort by the system to achieve equilibrium, although if changing outside circumstances and driving functions are available, this target equilibrium will be a moving one. In general, it is almost certainly reasonable to expect for the metropolitan region that some "sensible" equilibrium exists. The alternatives are exponential growth, collapse to an extreme configuration, or continuous oscillations. While none of these is impossible, they are not intuitively attractive. It therefore seems likely that one or more stable equilibria can be defined for most dynamic models. This is intuitively obvious for the Forrester model, given its output, and this equilibrium has been identified and analyzed. If for a dynamic model the equilibrium can be expressed analytically, it can also be explored for sensitivity to changes in parameters and to changes in policies. In principle there is no reason why such a model should not be "run backwards" so that policy variables would be set at a level required to maximize some welfare function. Such a backward-seeking model would be useful but of less general value than the dynamic programming model just described. It is very difficult to achieve because of the size, non-linearity, and possible existence of multiple optima in many large dynamic models.

Viewed in the light of the preceding paragraph, the distinction between static and dynamic models is not as great as might at first appear. An example of the blurring of this distinction might be found, for example, in the Lowry model of residential location, which is widely used and which has been developed in different directions by a number of workers. In the first place, while this model makes no direct claim to optimize, its equilibrium properties tend to suggest that some such process is at work at the behavioral level. More complex models containing market behavior and explicit optimization (such as the Herbert-Stevens model of residential location) probably produce similar results to Lowry's. It follows from these quasi-equilibrium properties that efforts to make the Lowry model dynamic can produce a succession of static equilibria which depend on changes in the over-all conditions which the model must meet. These successive equilibria may or may not preserve part of the previous decisions made in earlier runs of the model. Finally, Lowry himself saw a certain resemblance in successive iterations of the model which were needed for solution purposes, a rough analog of the time-phased physical development of the Pittsburgh region. On the basis of such resemblances there is indeed a substantial confusion in lay circles between iterations which are designed to solve the model at a single moment, and iterations over time which should more properly perhaps be called recursions.

There may be important statistical consequences of the similarity between static and dynamic models. In equilibrium the sub-areas of a metropolitan region would have constant composition, and either constant population or constant rates of growth. This would imply that in each area the internal forces leading to change in different directions would be exactly in balance. For a linear model, therefore, some combination of independent variables would be precisely collinear across all areas. Such a combination of equilibrating forces might be identifiable from cross-sectional rather than time series data. In this case, the principal roles of time series data would be quite different from their usual one. They would establish mainly the rate of adjustment to the equilibrium. It is important to note that if a growing organism like the metropolitan area has a set of internal forces of this kind which tend to lead to an equilibrium, then the multiplication of error in a projection tends to be minimized. The model is in a sense self-correcting, to the limits of the accuracy with which the relative importance of the various factors has been estimated. In rapidly growing areas or for slowly moving locators, the gap between equilibrium and the observed situation may be quite large and the errors may be not only substantial but biased. All of these remarks affect principally operational considerations and do not deny the basic relationship between static and dynamic models.

Mathematical neatness suggests that all variables in a model be treated symmetrically so that all equations for every locator look very much alike. If this kind of treatment is possible, a monolithic model may be developed which makes all types of projections at once. This is the case with the Forrester model of *Urban Dynamics*, the EMPIRIC model of the development of the Boston metropolitan area, and to an extent, the Lowry model. The difficulty with such monolithic concepts comes from a number of sources. First, for very large numbers of classes of locators or very fine small-area detail, the size of a unified model becomes outrageous, especially since the number of interactions tends to rise with the square of the number of variables. Second and more important, various different activities may have different modes of development which suggest the desirability of substantially different models linked together in some reasonable fashion.

A fair amount of experimentation has already been done with such linkages, and their character is clear on both practical and theoretical grounds. Essentially what may be expected to happen is that in studying any particular sub-system or large coordinated group of locators in the metropolitan region, the results of the activities of other locators are taken to be a part of the environment for the sub-system under study. At a later point in the process, the results of the activities of this locator become parts of the environment for other locators and influence their behavior. These interactions may be worked out by iteration at each single point in time, or they may be lagged and carried through successive steps in the recursion.

A major advantage of this type of subdivision of the problem is that highly diverse locator behaviors can be dealt with properly by distinctive models. It appears, for example, that a large part of retail trade location responds very rapidly to market conditions and is well represented by an equilibrium model. Residential locational choice is better represented by a model in which only a certain number of movers seek equilibrium and where this relocational behavior of a small proportion of the population represents a lagged dynamic element. Industrial location and the location of certain centralized services like banking are much slower to relocate than are households and require a still different model. One might expand this list very considerably, showing how public services of various types, household formation and dissolution, and various social phenomena each require their own type of model, and how these models may be operated in sequence and linked through a computerized data base which simulates the environment for all of them. Such a conception of modeling is flexible and easily amended. It is simple to define in principle and somewhat tedious to develop in practice or describe in detail.

A coordinated model system in which a variety of models interact with each other does not prejudge the issue of whether the total model will be dynamic or static. Such a group of models can be iterated to equilibrium and thus solved as a total system. Alternatively, the inclusion of a single dynamic model dynamizes the entire system.

I have not developed in any detail the concept and methodology of planning model design since this is substantially less mature than simulation. I define a planning model as a model which produces a plan, as in the case of a mathematical programming method, or which greatly assists a planner in producing a plan. Planning models are difficult to manage because of the large combinatorial searches which are involved, and it seems likely that this activity will best be left to a designer or decision-maker intervening in an interactive computerized system. This computerized system will have to have embedded in it simulation and evaluation models which can predict the results of the designers' efforts, but owing to the nature of the interactive process and the extent of the searches which are required, it seems likely that such simulations will have to be greatly condensed and simplified. In my view, we are perhaps in danger of proceeding too rapidly with interactive processes, without exploring the implications of the simplified simulations which they use.

The foregoing review has attempted to highlight some of the principal issues which surround the design of simulation models of urban metropolitan areas. These models currently exist in rather sophisticated forms and make heavy demands upon model design capabilities and upon computer power. Indeed it is altogether conceivable that the development of methods in this field will result in a substantial reduction of these demands at comparable levels of performance. The principal issues which I have discussed and which are subject to further research and investigation may be listed as follows:

1. The extension of substantive investigations into social and political spheres.
2. The investigation of elementary behavioral patterns coupled with an appropriate understanding both of disaggregation and of rules for aggregation or reaggregation.
3. An expanded understanding of the different structures, performance, characteristics, and uses of static and dynamic models.
4. The development of systems of linked models.
5. The development of planning models and interactive planning methods, together with the appropriately subordinate use of simulation as a part of these methods.

APPENDIX

Selected readings in urban simulation

Alonso, William. *Location and Land Use—Toward a General Theory of Land Rent.* Cambridge: Harvard University Press, 1964.

Berry, Brian J. L. Department of Geography Research Paper No. 85. *Commercial Structure and Commercial Blight.* Chicago: University of Chicago, 1963.

Chapin, F. Stuart, Thomas G. Donnelly, and Shirley F. Weiss. *A Probabilistic Model for Residential Growth.* Chapel Hill: University of North Carolina, Institute for Research in Social Science, in co-operation with U.S. Department of Commerce, Bureau of Public Roads, May 1964.

Chapin, F. Stuart, and Shirley F. Weiss. *Factors Influencing Land Development,* Chapel Hill: University of North Carolina, Institute for Research in Social Science, in co-operation with U.S. Department of Commerce, Bureau of Public Roads, August 1962.

———. *Some Input Refinements for a Residential Model.* Chapel Hill: University of North Carolina, Institute for Research in Social Science, in co-operation with U.S. Department of Commerce, Bureau of Public Roads, July 1965.

Forrester, Jay H., *Urban Dynamics,* M.I.T. Press, Cambridge, 1969.

Harris, Britton. "The Uses of Theory in the Simulation of Urban Phenomena," *Journal of the American Institute of Planners,* Vol. 32, September 1966.

———. *Highway Research Record No. 26: Land Use Forecasting Concepts.* Washington: National Academy of Sciences—National Research Council, Highway Research Board, 1966.

———. "Some Problems in the Theory of Intra-Urban Location," *Operations Research,* Vol. 9, September-October 1961.

———. "A Model of Locational Equilibrium for Retail Trade." Paper presented at a Seminar on Models of Land Use Development, Institute for Urban Studies, University of Pennsylvania, October 1964. Mimeo.

———. "Inventing the Future Metropolis." Paper prepared for the Catherine Bauer Wurster Memorial Public Lecture Series, sponsored by the Harvard Graduate School of Design and Massachusetts Institute of Technology. May 1966. Mimeo.

———. "The City of the Future: The Problem of Optimal Design." Paper presented at 13th Annual Meeting, Regional Science Association, St. Louis, Mo., November 1966. Mimeo.

Herbert, John, and Benjamin H. Stevens. "A Model for the Distribution of Residential Activities in Urban Areas," *Journal of Regional Science,* Vol. II, No. 2, 1960.

Journal of the American Institute of Planners. Special issues: *Urban Development Models: New Tools for Planning,* Vol. 31, May 1965; *Land Use and Traffic Models,* Vol. 25, May 1959.

Lowry, Ira S. *A Model of Metropolis.* Memorandum RM-4035-RC. Santa Monica: The RAND Corporation, August 1964.

———. *Seven Models of Urban Development: A Structural Comparison.* P3673. Santa Monica: The RAND Corp., September 1967.

Muth, Richard F. "The Spatial Structure of the Housing Market," *Papers and Proceedings of the Regional Science Association,* Vol. 7, 1961.

Orcutt, Guy, John Korbel, Alice M. Rivlin, and Martin Greenberger. *A Microanalysis of Socio-Economic Systems: A Simulation Study.* New York: Harper, 1961.

Seidman, David R. *A Linear Interaction Model for Manufacturing Location,* Penn-Jersey Transportation Study. Philadelphia: Delaware Valley Regional Planning Commission, 1964.

Wingo, Lowdon, Jr. *Transportation and Urban Land.* Washington: Resources for the Future, Inc., 1961.

The protection of privacy and security in criminal offender record information systems

by STANLEY ROTHMAN

Consultant
Manhattan Beach, California

INTRODUCTION

In this paper we will single out those aspects of the problem of protecting privacy and security in information systems that are special to law enforcement.

FEDERAL-STATE RELATIONSHIP

The National Crime and Information Center, which extends from the FBI to the state, county, and city level, has been expanded to contain and exchange criminal histories. The rules under which state and local governments participate in this system are under debate, a debate that may extend to a law suit by the State of Colorado against the FBI. The substance of the conflict is the ruling that any computer participating in this on-line exchange of criminal histories must either be dedicated to law enforcement or under the management control of law enforcement. The significance of this is as follows:

a. Neither the FBI nor the Federal Government control local law enforcement.
b. There are at least a dozen states that can only afford a shared service bureau installation.
c. Management control of non-enforcement reccords, such as welfare or health, by law enforcement will cause another debate, a very loaded one.

Other technical requirements are dedicated communications lines and non-dial-up terminals.

PROJECT SEARCH

The Law Enforcement Assistance Act has for some time, thru Project SEARCH, sponsored development of technology, a model state act and administrative regulations for the protection of privacy and security in this exchange of criminal histories. However, this work is strictly advisory. All fifty states now participate in the work, but there is guarantee of neither unanimity nor state legislative approval of the results.

LAW ENFORCEMENT

Law Enforcement is the principal participant in this system to date. Thus, the system operates twenty-four hours per day, seven days per week. Eventually the courts, prosecutors' offices, probation, parole, prisons, and the entire criminal justice system will participate.

The information system competence of law enforcement is highly variable. They are in general too dependent on the equipment manufacturers. Their information systems have to serve many other purposes than enforcement, such as credit, military clearance, and licensing. They have some experience in handling "need-to-know" type restrictions for vice records, but the whole idea of restricting access to arrest records that do not have convictions will take some getting used to.

Law Enforcement must manage personnel within Civil Service regulations. Thus, screening out people with a criminal record, criminals in the family, or firing an employee for violating administrative security regulations is difficult. Similarly the use of the polygraph as a control is subject to fifty different sets of state laws.

A large number of law enforcement installations still operate manual files and these must be protected, perhaps even more stringently than automated ones. This is because they may have a terminal that receives criminal history information even if they do not have a computer that is linked to the network.

One further requirement that is not unique but is important is the facility for research in criminal records. Longitudinal studies have to be done of the criminals in their progress thru the criminal justice system. These studies require added protections because of their potential for violating the privacy of the criminal.

THE THREAT

Within this context of ambiguity and good intentions there are the threats to law enforcement information systems that are very real and very specific. They are:

a. The anarchist who wants to disrupt, destroy or embarrass the system;
b. The criminal who would like to remove a file or query the file of another criminal;
c. The private detective, bank officer, newspaper reporter, or employer who would like to check for a criminal history; and
d. Civil disorder.

The access can be gained either with some difficulty from outside the system or thru misuse of people with legitimate access. All of these threats have taken place at one time or another. The most common threat is the bribery of system employees and police officers. The technical threat has been documented elsewhere and is little different than the technical threat to any computer—communications system. One of the differences is the extent to which it is worthwhile to protect against wire-tapping and electromagnetic radiation. Until an organized crime intelligence exchange is automated this added expense is not justified. This is not so much a judgment on the cost of the protections as it is an estimate of the small value of most of this traffic.

With the exception of juvenile records, most of the information in criminal records is a matter of public record. It is uniformly agreed that errors in these records should be corrected. However, since these records are widely disseminated, the dissemination records must be maintained to direct the distribution of the error corrections. This by itself is a considerable task. An unusual requirement is that under some circumstances all evidence that a criminal record existed must be removed.

CONCLUSION

There are several things that can be said about the solution to these problems. First of all, the achievement of a commercially available secure operating system is vital to resolving the debate about the relative security of shared versus dedicated installations. I suggest that the computer manufacturers pay attention to the requirement for a secure operating system. There is every evidence that these federally sponsored non-military agencies will unite at the Federal level to produce binding procurement specifications that could be influential.

While many manufacturers have been working on absolute identification of terminal users thru voice, fingerprint or handwriting recognition, I would like to underscore the importance of success here. It is the key to the control of one-man remote terminals.

A problem area that has so many requirements, purposes, decision makers, and an incomplete technology—the technology of computer protection—ends up with procedural protections. These are inherently weak because they depend upon human diligence. It is for this reason that the management control of shared installations that contain criminal records requires the added protection of the discipline that is traditional with the police.

Lastly, while Law Enforcement has been particularly farsighted by working on this problem well in advance of an uproar like that created by the proposal for a National Data Center, the achievement of a secure, nation-wide criminal history exchange that protects privacy could well take a good deal more time and trouble. In part this is because of the absence of concrete cost trade-offs studies that tell us how much reduction in risk our security measures buy. However, even more important is the fact that such an exchange requires a uniformity of state laws governing the use of criminal histories. Such uniformity will be difficult to achieve.

Security of information processing—Implications from social research*

by ROBERT F. BORUCH

Northwestern University
Evanston, Illinois

INTRODUCTION

Many social research programs are characterized by a stringent requirement that identifiable data collected on the subjects of research be kept confidential. This requirement, coupled with the increasing number of sensitive, sometimes controversial research efforts, has stimulated social scientists' interest in legal, administrative, and technical methods for assuring that confidentiality is maintained. We concern ourselves primarily with the technical methods in this paper, treating "security" as a partial operationalization of the notion of confidentiality.**

Specifically, we should like to sketch those problems met in social research which are relevant to security-oriented activities in information processing. In the following remarks, some of the distinctive features and needs of large-scale social research are outlined. Then, the research design, data collection, maintenance and dissemination stages of the research system are examined to discover how the interests of social research and those of security-oriented information processing might intersect vis-a-vis the problem of assuring confidentiality.

THE CHARACTER OF SOME SOCIAL RESEARCH PROGRAMS

Maintaining confidentiality and security of data are likely to be important objectives in a variety of social research efforts. In the section, examples of these are furnished and the factors which appear to be important in distinguishing research archives from other kinds of information systems are described briefly.

Focuses of the research

In order to establish a manageable topic area, suppose we restrict attention to large-scale social research which results in a computerized information system containing data on identified research subjects. Some form of identifiers (e.g., names and addresses) are essential when individual subjects must be tracked over time to investigate biological and social development, to appraise the cumulative impact of drugs or alcohol abuse, etc. These so-called "longitudinal studies" are frequently conducted, and although many are quite small, some involve repeated in-depth measures on over 100,000 individuals over a 10 or 20 year period.

The research topics which can be expected to generate some concern about privacy, confidentiality, and security cut across all the social sciences. In political science, for example, whether an individual voted or not is frequently a provocative topic for inquiry and a negative response usually constitutes "sensitive" information. Human factors psychologists, often involved in accident research, focus on seat-belt wearing behavior; in some highway surveys, spot checks are made of drivers' alcohol use. Each type of information may have a stigmatizing character. Epidemiologists, of course,

* Work on this topic has been supported by NSF Grants GS320-73X and GI29843. Naturally, the views expressed here do not necessarily reflect those of the sponsoring agency. Some of the observations made in this paper are an extension of earlier research reports, notably (1).

** Confidentiality here refers to the status of information, a condition under which access is formally restricted to certain agencies or individuals. Security refers to the administrative, technical, and legal devices used to assure that the formal restrictions are met; i.e., security is an operational definition of the concept of confidentiality.

frequently need to acquire longitudinal data on incidence and spread of venereal disease, on illegal abortion, and on other socio-biological deviations from the norm. Social psychology, traditionally concerned with relatively innocuous laboratory experiments, has become associated with research on white-collar crime, on mob violence, and on helping behavior in critical situations (e.g., bystandar apathy to a streetcorner mugging). Large-scale research in economics and law has, in recent years, accumulated much longitudinal data on individual's spending behavior, deviations between actual and reported taxable income and other sensitive topics. (For references to work in each area, see Reference 1.)

In the past, confidentiality has not been so crucial and generalized a concern because the size of the research efforts had been small and visibility of the studies low. Perhaps more importantly, the academic orientations seemed to have been associated with relatively innocuous data on anonymous individuals or subjects tracked over very short time intervals. During the past five years, the size and visibility of social research projects such as those described above has increased dramatically, particularly in the policy research and evaluation areas.[2,3] As in commercial data collection activities, accidental disclosure and deliberate penetration of research files can have serious consequences: research subjects may be embarrassed or harassed and the research programs would undoubtedly suffer. Although the empirical risks here are sometimes no better documented or appraised than risks in commercial data collection enterprises seem to be, the issue is serious enough for both Federal and private grant agencies to develop guidelines on collection and maintenance of identifiable data on individuals (see references in Reference 2). The social researcher's interest in establishing the security of information stems from increased visibility of research, from these formal legal requirements, and from the ethics and the realities of research. Our ability to collect data will suffer considerably unless we conscientiously and conscionably recognize the need for security.

Archival data: Functional distinctions relevant to security

How might we describe the functional character of social research data archives and those features which appear to be important for the sake of security? As a first approximation, we might consider a rough continuum of computerized data banks which contain personal records, defining the continuum such that one end represents an auditing function and the other represents a research function. Personnel records and intelligence systems typify the first extreme, where each identifiable record serves as a basis for making evaluative judgments about the individual on whom the record is kept, and for taking direct and personal action which directly affects the individual.

The research-oriented systems generally serve not as a vehicle for decision and action about an individual, but for appraising the group's condition with respect to some social theory or with respect to the effectiveness of a program with which the group is involved. The American Council on Education's Higher Education Data Bank,[4] and Project Talent[2] exemplify this activity. Each collects identifiable data on thousands of students annually. Most of the data are innocuous by any standards, but some pertain to campus protest activity, alcohol use or other sensitive behavior. Identifiers serve as an accounting device, and the data are not meant for use as a basis for evaluative decisions about individuals.

The functional distinction—audit versus research—has some rather important implications for minimizing the likelihood of disclosure or the utility of data should the data be deliberately appropriated for nonresearch purposes. Identifiers, even if collected, do not need to be as accessible as statistical data for research purposes. Special strategies for separate handling of identifiers and statistical records can be developed and have been used to minimize risk of disclosure (see Intrasystem Linkages, below). Statistical records in audit systems usually must be quite accurate, but in the social research systems, imperfections generated by the method of data collection are recognized and estimated, not for the individual, but for the group as a whole. In fact, to undermine the utility of individual records, without jeopardizing the integrity of the total data seriously, random error whose parameters are known can be inoculated into the data.* This strategy, evidently inappropriate for commercial record systems, seems to hold some promise in research concerning topics such as use of contraceptives and illegal abortion,[5]

* Some research designs can be set up such that each respondent injects his response with random error in a manner prescribed by the researcher. For example, in a question requiring a yes-no response, the researcher might instruct the subject to roll a die and to lie if a "1" shows and to tell the truth if 2, 3, or 6 shows. The known likelihood of false positive and negative responses in the paradigm can be used to obtain unbiased estimates of parameters in data analysis. The presence of randomized error in the record system would presumably reduce embarrassment, and threats of unauthorized or legal disclosure, since individual records cannot be used for unambiguous judgments about individuals on whom records are kept.

Linkage problems also differ a bit depending on function. In the research systems, one often wishes to merge identifiable data collected by different agencies. Unlike merges in many audit systems, the separate agencies each may have their own rules and practices regarding disclosure of individual records but may be willing to share data if rules about confidentiality are not compromised. The researcher must then devise special strategies to link data without breaching these rules and without compromising the promises of confidentiality made to individuals on whom records are kept. Specialized methods have been developed (see remarks below on Intersystem Linkage) but more work needs to be done.

The legal status of information in social research also differs from data in the audit system. In some states, socio-medical research records, some educational and psychological records are protected from even legal interrogation by a testimonial privilege. More often, however, they are not so protected and some mechanisms have been devised to undermine the data's legal utility or to minimize its legal accessibility. The inoculation of random errors probably meets the first objective; specialized froms of data linkage and maintenance (Intrasystem Linkages, below) help to meet the second. These legal differences are related to security needs in general, and since processing is typically conducted with computing machinery some particular features of information processing technology may also be relevant here.

Each of these differences imply some of the specialized needs of the research data archive in contrast to the audit information system. In the next sections of this paper, the collection, processing, and maintenance stages of the research system are described in a bit more detail and linked to methods for assuring security of data.

DATA COLLECTION

In the simplest case, data are elicited by the researcher and an individual's response transmitted back to the researcher through various intermediary groups. The intermediaries often include local administrators, staff members of scanning/mark sense processing units, and key punch operators as well as the researcher's personal representatives. For the sake of security, many social researchers are attempting to reduce the possibility of disclosure to intermediaries, particularly by reducing the number of intermediate stages between eliciting information and the provision of response.

Questionnaire surveys

In order to eliminate the possibility of disclosure during survey administration, some plans require the respondents to put the completed questionnaires into locked and addressed boxes which would be sent directly to the data processor. In some cases, representatives of various interested and disinterested groups can and do monitor the collecting, packaging, and mailing of completed questionnaires.

Even more simply, questionnaires or interview documents have been designed so that one section, containing identification and code number, can be detached from the other, containing responses and an identical code number. Either the respondent at the site of the survey or the researcher at later stages of the survey process can actually separate the two components of information. The identifying information can then be held by the respondent or by a monitoring agency (e.g., group of respondents or representatives of the host agency) and submitted to the researcher after the statistical information is compiled. The code numbers permit later linkage of statistical data with information collected later in the research process.

Rather than require individuals to respond directly on a questionnaire, some researchers are making more use of perforated, but otherwise standard EAM cards as a vehicle for recording data. In requiring that the respondent merely punch his responses out on the card and return it by mail, any intermediate handling of identified records is reduced. And, we can couple this strategy with the use of nominal or numeric aliases to further enhance security. The principal problems with this approach seem to be subjects' reaction to the cards and limitations of the card format on permissible response options. Human engineering studies would probably help to ameliorate some of these problems.

Remote terminals

One idea which seems to have some merit involves the use of remote terminals as a kind of voting booth for repeated surveys of certain groups of individuals. That is, rather than have respondents furnish data via questionnaire or telephone, we might require that they do so through "social reporting units" in which opinions and self-descriptions can be input directly to storage by an individual. Remote input devices might be particularly useful in organizational settings where continuous monitoring of individual's attitudes, activities, expectations, and status are essential for research on the effects of policy changes or of organizational innovations.

The voting booth or other remote input methods might, for example, be applied usefully to public housing appraisal where good data on resident's status is essential to economic studies.[6,7] Othera pplications may include welfare recipient's reporting, transportation depot surveys, or surveys of any well-defined group (e.g., hospital, military, prison or student groups), whose members can provide useful input data to the social research reservoir. In many such reporting systems, a guarantee of anonymity is necessary for honest and continued reporting; however, tracking the development of individuals is also a frequent requirement. These two needs suggest creation of systems in which the technically *un*sophisticated respondent can make inputs easily and without being jeopardized by the opinion or factual information he offers. The numeric alias or password systems already developed appear to be relevant here. Some are persuasively secure, e.g., permitting the respondent to form his own transform of a random number of identifiers supplied by the computer. The human factors problems in getting people to use and to adhere to their personal, private transforms will probably outweigh the technical problems in implementing such a system, but these do not seem to be intractable.

DOCUMENT PROCESSING

Anonymous reporting, responding under alias identifiers, and using specially constructed questionnaires (or having respondents inoculate their response with random error), usually minimize if not eliminate the likelihood of unauthorized disclosure at later stages in the research system, including document processing. But these strategies may be inappropriate or too expensive for particular kinds of research. Very large and very expensive field experiments, for example, are an important means of evaluating economic and other governmental programs; intensive and long term longitudinal surveys of small samples contribute to our knowledge of human development.[3,8,9,10] Both kinds of studies typically require exhaustive cross-checking capabilities, very complex merge operations, and other activities which appear to justify the joint processing of statistical and identifying information. The use of aliases in these cases may be completely inappropriate and the use of specially constructed documents may make cross-checking the validity or completeness of response very expensive.

In these circumstances, the social researcher usually meets several problems. For one thing, document processing agencies often have neither written policies nor formal administrative regulations regarding the treatment of sensitive data. Similarly, the paucity of information on the establishment of and adherence to codes of ethics in the document processing industry is serious concern to many researchers; since the document processing is frequently (perhaps necessarily) tied to computer operations, the concerns apply to this area as well.

When no administrative or ethical codes are espoused by the service agency that the researcher must employ, it may become necessary for the researcher and the data processor to reach some formal contractual agreement on the treatment of data. At a minimum, such agreements should require that identifying data and response data be separated at an early stage, that the documents be destroyed soon after processing, and that the responsibilities and consequences of negligence on the part of the service agency be carefully defined. At present, insuring that such a prescription is adequate can be difficult because legal precedents and specification of negligence and liability in a document or data-processing environment have not been fully established. The current explorations of these legal problems may clarify the situation (see references in References 2 and 11).

MAINTENANCE AND DATA LINKAGE

When identifying information must be collected with data, the device most frequently used by social researchers for minimizing accidental disclosure or deliberate interrogation of identifiable records is physical separation of identifiers and statistical data. Each separated file usually contains code numbers which permit later merging operations and the identifier file is often kept in vault storage. A few social research agencies have applied some of the Department of Defense administrative and mechanical requirements for security, and the agencies often require computer service groups with which they deal to use the same regulations where feasible.[4,11]

More elaborate schemes for minimizing the likelihood of disclosure have been developed and are being used. Many of these strategies can be divided into three groups depending on the purpose of maintaining identifiers: schemes for intrasystem linkages, for intersystem linkages, and for combined audit-research systems.

Intrasystem linkages

Intrasystem linkages refer to a single agency's collecting and merging data on the same sample of individuals over an extended time period. In longitudinal studies of students' political activism, for example, data are frequently collected in identified form. It is reasonable to expect that nonresearchers may be interested in examining identifiable data. The researcher with no legal testimonial privilege (i.e., without the ability to resist subpeona), would normally like to minimize or eliminate the possibility of disclosing sensitive data to even legal authorities when he has promised confidentiality to his respondents.

An interesting operational resolution of this problem is the American Council on Education's LINK FILE SYSTEM.[4,11] The strategy was developed to assure the confidentiality of longitudinal data on college students, data which includes limited but identifiable information on disruptive campus protest activities. It works in the following way.

After identified questionnaires are returned by students, the researchers split the information into two segments. The first contains statistical data with one of arbitrary numerical codes attached to each record; the second contains students' names and addresses linked to a second set of code numbers. A third file matches the first and second set of numerical identifiers (aliases). This code linkage is kept in a foreign country with an agency contracted to maintain the linkage for later data merges; the agency is also required by the contract *not* to return the linkage to the researchers under any circumstances. In followup studies, the researcher's name and address file is used to distribute questionnaires. The associated numerical aliases are substituted for names during document processing and this file is then shipped to the contract agency. The agency replaces the numerical identifiers in this file with the first set of identifiers, using its code dictionary. Then, this follow-up file is returned to the research agency for merging follow-up data with the original data, using the numerical identifiers common to both files (i.e., the first set of arbitrary numerical identifiers).

The system is certainly flawed in that it can be undermined in some cases by the research staff, by the agency holding the code linkage file, and by legal agencies with international ties (see the Hoffman and Turn critiques described in Reference 11). But it is a useful prototype which may help us learn a bit more about how to design and implement a system which will assist in protecting social research data. It does provide a concrete target for the check list strategy given by Peterson and Turn,[12] to determine susceptibility of the data to legal interrogation and corruptibility of the system by its creators as well as by outside agencies. The difficulty of using encoded identifiers (arbitrary identification numbers) in physical protection for files and of protecting against indirect disclosure overlap considerably with problems in intersystem linkages which we consider next.*

Intersystem linkages

Intersystem linkage refers to the researcher's merging his own identifiable research records with records maintained under other auspices. As an example, consider the (true) example of an economist who obtains data on spending behavior and wishes to correlate these with items from income tax returns. The linkage of both sets of records raises difficulties of two kinds. On one hand, the researcher's provision of identifiable information to the IRS for merge purposes may violate his promise to his respondents assuring the confidentiality of the data. On the other hand, the researcher cannot obtain identifiable data from the IRS for merging because IRS regulations generally prohibit such disclosure.

The so-called insulated linkage process for merging data is an illustrative resolution of these two problems. To link the files, the researcher first cryptographically encodes all statistical data in his own records. He then supplies the joint records (encoded statistical data coupled with identifiers), to the other archival agency. The latter then merges its own files with the researcher's file, basing the merge on the identifiers appearing in both files. When the merge is complete, identifiers are deleted and the resultant file, consisting of unidentified statistical records from both agencies, is then returned to the researcher. This system has been used in actual merge operations with some success and is one of a general class of strategies for linking data under security restrictions.[11]

Again, a linkage strategy of this sort can sometimes be rather vulnerable, and additional mechanisms must be invented to minimize deliberate efforts to interrogate identifiable data in either file. To corrupt the system, the researcher could, for example, encode a duplicate set of identifiers in his file, allowing identifiers to masquerade as statistical data. Presumably this strategy can be rendered useless by having the archival agency not only merge the data but also summarize it. The provision of summary data then may permit only indirect disclosure efforts by the researcher. But if the researcher uses a very simple encryption scheme, such as systematically substituting one character for another in the records, the archival agency may be able to penetrate

* Indirect disclosure involves using a twenty questions strategy to deduce new information about an identified individual when the interrogator has statistical as well as identifiable records on the individual.

the substitution scheme and in fact examine the researcher's identifiable records.

In both the intrasystem and intersystem linkages, social researchers need more guidance on appraising the vulnerability of the strategies. Aside from making more thorough appraisals as outlined, for example by Peterson and Turn,[12] we should obtain better insights into more systematic ways of detecting and inhibiting the likelihood of indirect disclosure, and the utility of cryptographic encoding in these applications.

Combined audit-research systems

Some organizations, governmental ones especially, have both audit and research missions and the information maintained in their computers reflects this dual objective. Trust may be a reasonable basis for assuring that researchers will not improperly explore identifiable administrative records or that nonresearchers will not interrogate identifiable research records. More formal restrictions on access and disclosure may be warranted, however, particularly where the data vary considerably in sensitivity, and administrative or personnel monitoring procedures are difficult to implement.

Some of the researcher's needs here can be characterized as having two dimensions. On one hand, he has some need for a flexible, hierarchical system of protection for his own data which can be tailored to the pyramidal nature of its sensitivity. Innocuous and public data might then be kept secure with the cheapest form of protection possible, e.g., existing administrative checks on personnel and the physical plant. More sensitive information such as sources of income, psychiatric and hospitalization records, personal habits and beliefs would justify more secure (and presumably more expensive) mechanisms including those reviewed by Hoffman,[14] say, in his state-of-the-art survey. Some flexibility is essential if the researcher is to keep pace with both changing public opinions regarding the sensitivity of stored information and the changing substance of research. These requirements may be met with the development of hardware modules or micro-coded instructional sets which the researcher himself can use as building blocks for made-to-order protection of data with different levels of sensitivity.

On the other hand, the audit portion of a combined audit-research system may warrant authority hierarchies for access to data which are geared to administrative and researchers' needs. Normally the social researcher wishes to meet his research objectives without incurring the responsibility or liabilities associated with access to joint information and without forcing a compromise of the original conditions (e.g., a promise of confidentiality) under which information was originally supplied to an audit agency. The Shared File System (APL) developed by David Booth[15] appears to have some relevance to this problem; it involves the use of access authorization codes associated with particular primitive (and unmodifiable) commands and particular roles. Presumably, research needs can be accommodated well by tailoring the system so that the researcher can operate with restricted functions in restricted work spaces and arrays, while locked out of his administrative or research colleagues' work spaces, and unable to examine or modify other functions and files stored in the same equipment.

DISCUSSION: POTENTIAL USE OF A DATA BANK REGISTRY AND DEVELOPMENT AGENCY

A paper as brief as this one must be cannot hope to give a detailed appraisal of the social scientists' needs in their efforts to maintain the confidentiality and security of the data they maintain. As a framework for summarizing those needs, suppose we consider the current proposals for a national registry of computerized data banks. The proposals are in the interest of developing mechanisms for solving problems in the security area and they may be helpful at the design as well as implementation stages of social research.

It has been suggested that such a registry, coupled with a development agency, be created for the purpose of documenting the nature of computerized information systems, the kinds of personal data maintained in such systems, and the rules and practices which pertain to storage of data. Alan Westin's proposed "data bank on data banks,"[16] John Kemeny's plan for a National Computer Development Agency,[17] and other suggestions for monitoring large-scale data collection[8,18,19] seem to imply documentation functions of this sort. We can anticipate that such plans, if implemented, will be of considerable interest and use to social researchers, especially if they include the kinds of information listed below.

POLICY AND PRACTICES IN DATA COLLECTION

Given the diversity of social research programs, no single policy or managerial practice is likely to satisfy all public and private requirements for assuring confidentiality of data. Statistical methods for minimizing likelihood of identification, legal constraints against access as disclosure and administrative methods for assuring confidentiality have been developed, but they have been organized and appraised in only a few instances.[2,5,20] Regrettably, these strategies have not been tied well to more computer-bound technical devices such as those described by Hoffman,[14] Peterson and Turn,[12] and Goodfellow.[21] An agency with an information clearinghouse function, coupled with a development mission, would be quite helpful in documenting, consolidating, and organizing information in the following categories.

Legal solutions

Local, state, and Federal statutes relevant to privacy and confidentiality of data; court precedents, administrative regulary powers; empirical data on problems in enforcement of codes, and adherence to guidelines furnished by government agencies to social researchers regarding rights of privacy and conditions of disclosure.

Administrative approaches

Link file systems,[4] insulated data banks,[11] and other similar strategies for eliciting and merging sensitive data; vulnerability, utility, and frequency of the strategy's use; cost data.

Statistical/Mathematical Solutions

Documentation on applications of error inoculation[5] and other approaches to depreciating probability of indirect disclosure;[13,20] costs and benefits of applications.

Technical mechanisms

Types of cryptographic encoding appropriate for computer applications; their cost and vulnerability; catalogs or listings of hardware and software security devices; possible relevance of new devices to specialized research needs (such as remote terminal application mentioned earlier; see also Reference 22).

Empirical studies

There is some real value in consolidation of data on people's resistance to data collection and to social research. Complaints about the collection of information and against organizational disclosure practices, concerns about the magnitude of data maintained, etc., need to be well-documented. Although some empirical data exist, there is currently no single source on which the researcher may draw to establish the likelihood of privacy problems in the conduct of his research and to anticipate the costs of resolving them.

In many cases, questions can be phrased to minimize embarrassment and/or threats of sociolegal action against a respondent. Some of the relevant strategies—elimination or generalization of the inquiry, approximations to direct questions—are fairly well documented.[2] Small "item pools" or computerized retrieval systems containing questions which pertain to the same general behavior, but with varying levels of sensitivity and intrusiveness do exist. But data on both strategies and item pools are widely dispersed. There is still a great need for large, accessible item pools which have been tested for objectionability, intrusiveness, and susceptibility to error.

Validity appraisals and secondary analysis

Frequently, social researchers elicit anonymous information from previously identified samples or require research subjects to use an alphanumeric alias (in short term longitudinal studies), so as to minimize if not eliminate any risks that data will be used for nonresearch purposes. An information registry would be of considerable help in appraising validity of sampling and credibility of reporting in such efforts. Suppose, for example, a medical sociologist, who usually has no testimonial privilege for the data he collects, relies on mailed or telephoned responses to his questionnaire on illegal methadone use. He might encourage the use of aliases to assure that his data are not appropriated (legally or otherwise) for harassment of his subjects, but he still needs to anticipate the redundancy of his data, and to appraise its validity since he does depend on voluntary responses. The researcher could do so if a data bank register furnished information about the existence of medical records, census data, police intelligence systems, etc., which contained relevant statistical data on the population from which subjects were sampled. And if identifiers were actually obtained he could merge his own data with existing files without violating access restrictions using some special administrative strategies which might also be documented in the same registry.

SUMMARY

The objective of maintaining security of social research data is an operationalization of the concept of "confidentiality" in social research. The problems in meeting the objective depend on where the research falls on a hypothetical audit-research continuum for the data, on the kinds of process being used to elicit the data, and on the level of identifiability of records necessary in the research. Major differences between audit and social research approaches to security problems stem from the social researcher's infrequent need to maintain joint identifying and statistical records, and the opportunity to use modified (alias) identifiers and modified response data (i.e., inoculated with random error in a controlled process).

Aside from benefiting from systematic appraisal methods such as those described by Peterson and Turn,[12] social researchers might do well to capitalize on other research efforts connected with security in information processing. Linkage systems and similar devices mentioned earlier depend very much on encryption schemes for assuring integrity of the system. The encryption transforms used in the examples cited have been limited to simple substitution of one character for another or simple linear transforms of original numerical characters. Perhaps certain kinds of transposition or additive transforms, as yet unfamiliar to the social scientist, can be adapted to this kind of problem to assure greater security. Certainly, the development of algorithms which help in checking whether indirect disclosure is possible or likely would be well received by managers of the research data banks. Translating the structure of data sets into simple algebraic equations is a skill which is usually beyond the social scientists' own expertise. Judging from Fellegi's[13] work and current activities by Turn,[23] such algorithms are likely to require a great deal of techinal attention to efficiency, to heuristic alternatives to searching large sets of equations (data sets), to determining the likelihood of indirect disclosure, tasks in which the social must be educated by the computer technologist.

Certainly, if proposals for national data registries and development centers are implemented, social scientists will have the opportunity to reduce redundancy in collection and maintenance of identifiable data. A centralized information source may help to stimulate more interest and expertise in technical solutions to problems in this area. Since most social research involves data which are heterogeneous with respect to their sensitivity and publicity, the researcher will benefit most from technological developments which associate more protection with increasing levels of sensitivity, and authority access designs which recognize these levels.

REFERENCES

1 R F BORUCH
An annotated bibliography of randomized field experiments in policy research
Background paper for Social Science Research Council's Committee on Experimentation Northwestern University 1972

2 R F BORUCH
Maintaining confidentiality in educational research: A systemic analysis
American Psychologist 1971 26 pp 413-430

3 T K GLENNAN
Using experiments for social research and planning
Monthly Labor Review February 1972

4 A W ASTIN R F BORUCH
A "link" file system for assuring confidentiality of research data in longitudinal studies
American Educational Research Journal 1970 7 pp 615-624

5 R F BORUCH
Administrative, statistical, and legal solutions to the problem of assuring confidentiality in social research
Paper presented at Statistics Department Colloquium University of Chicago 1972

6 J ROTHENBERG
Urban economics
In Nancy D Ruggles (Ed) Economics: Report of the behavioral and social science survey (NAS and SSRC) N J Prentice-Hall 1970

7 H BLACK E SHAW
Detroit's social data bank
In A F Westin Information technology in a democracy Cambridge Harvard University Press 1971

8 E B SHELDON
Social reporting for the 1970's
Chapter 7 Report of the President's Commission on Federal Statistics Washington DC US Government Printing Office 1971

9 W D WALL H L WILLIAMS
Longitudinal studies in the social sciences
London Heinemann 1970

10 D T CAMPBELL
AdministratiJe experimentation, institutional records, and nonreactive measures
In W M Evan (Ed) Organizational experiments Laboratory and field research New York Harper and Row, 1971

11 R F BORUCH
Strategies for eliciting and merging confidential social research data
Policy Sciences September 1972 (in press)

12 H E PETERSEN R TURN
System implications of information privacy
Proceedings of the 1967 Spring Joint Computer Conference American Federation of Information Processing Societies 1967

13 I P FELLEGI
Question of statistical confidentiality
Journal of the American Statistical Association 1972 67 pp 7-18

14 L J HOFFMAN
Computers and privacy: A survey
Computing Surveys 1969 1 pp 84-103

15 D F BOOTH
File security for a shared file, remote terminal system
Paper presented at the Conference on Computers, Privacy, and Freedom of Information (Mimeo) Queen's University 1970

16 A F WESTIN
Civil liberties and computerized data systems

In Martin Greenberger (Ed) Computers, communications, and the public interest Baltimore The Johns Hopkins University Press 1971

17 M GREENBERGER (Ed)
Computers, communications, and the public interest
Baltimore The Johns Hopkins University Press 1971

18 President's Commission on Federal Statistics
Report of the President's Commission
Washington DC US Government Printing Office 1971

19 G B F NIBLETT
Digital information and the privacy problem
Paris Organization for Economic Cooperation and Development 1971

20 M H HANSEN
Insuring confidentiality of individual records in data storage and retrieval for statistical purposes
Proceedings of the 1971 Fall Joint Computer Conference American Federation of Information Processing Societies 1971

21 B B GOODFELLOW
Projections of the impact of technology on the development of large data base information systems
Position paper presented at the Conference on Computers: Privacy and freedom of information Queens University Kingston (Canada) May 21-24 1971

22 N M BRADBURN
Survey research in public opinion polling with the information utility—promises and problems
In H Sackman and N Nie The information utility and social choice Montvale (New Jersey) AFIPS Press 1970

23 R TURN N Z SHAPIRO
Privacy and security in data banks: Measures of effectiveness, costs, and protector-intruder interaction
Proceedings of the 1972 Fall Joint Computer Conference American Federation of Information Processing Societies 1972

Privacy and security in databank systems— Measures of effectiveness, costs, and protector-intruder interactions*

by REIN TURN and NORMAN Z. SHAPIRO

The Rand Corporation
Santa Monica, California

INTRODUCTION

The nearly seven years of concern with data privacy and security in computerized information systems have produced a variety of hardware and software techniques for protecting sensitive information against unauthorized access or modification.[1–7] However, systematic procedures for cost-effective implementation of these safeguards are still lacking.

The data security design and implementation process will remain more art than science until adequate theoretical foundations are laid and analytical tools developed for a "data security engineering" discipline. Needed in particular are measures for evaluating the effectiveness of data security techniques in various threat and implementation environments; methods for estimating the costs of implementing the safeguards in various classes of information systems; and tradeoff relationships between these and other relevant variables. Equally important is the ability to estimate potential losses.

This paper strives to contribute to the formulation of data security engineering in the areas of personal information databank systems: a model of the personal information databank system is presented; the nature of the interactions of the databank security protector with potential intruders is explored; and the amount of security and implementation costs associated with several classes of data security techniques are discussed.

THE DATABANK SYSTEM

The term *databank* implies a centralized collection of data to which a number of users have access. A computerized *databank system* consists of the data files, the associated computer facility (processors, storage devices, terminals, communication links, programs and operating personnel), a management structure, and assorted "interested parties."

Structure

If the function of a databank system is to collect, store, retrieve, process, and disseminate *personal data* on individuals (or organizations), the databank system includes the following elements:

- *Subject,* a person or an organization about whom data are stored in the databank system. He may have provided the data voluntarily, in a quasi-mandatory fashion to obtain benefits or privileges, or as required by law. Data on him may also have been collected without his knowledge or consent.
- *Controller,* an agency or institution (public or private) with authority over the databank system and its operations. The controller authorizes the establishment of the databank system, specifies the population of subjects and type of data collected, and establishes policies for the use, dissemination, disclosure, and protection.
- *Custodian,* the agency and its personnel in physical possession of the data files, charged with the operation of the databank system, and responsible for enforcing the policies established by the controller.
- *Collector,* the agency and personnel who collect the data and transmit it to the custodian.

* The research reported in this paper was supported by the National Science Foundation Grant No. GI-29943. Any views or conclusions contained in this paper should not be interpreted as representing the official position or policy of the National Science Foundation or The Rand Corporation.

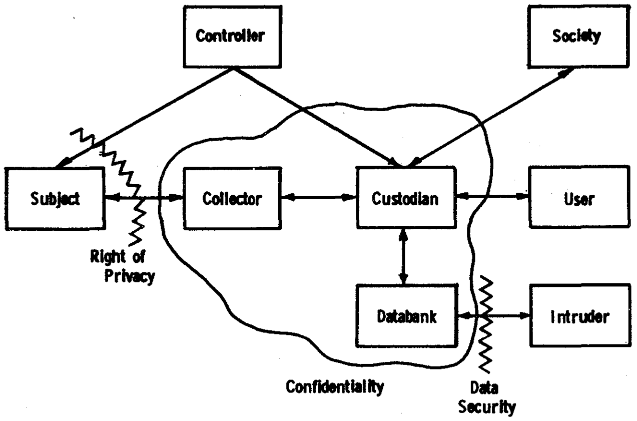


Figure 1—The Databank System

- *User*, a person or agency authorized by the controller or the custodian to utilize specified subsets of data for specified purposes, subject to the disclosure and dissemination policies of the databank system.

Other parties interested in the data and its uses include:

- *Intruder*, a person or agency either deliberately attempting to gain unauthorized access to the databank system or making unauthorized use of the data normally available to him as an authorized user, or accidentally doing so.
- *Society*, the population within which the subjects have rights and obligations, and whose welfare also affects the welfare of the subjects. Large classes of databank systems are needed to support studies of the society, and administer and assess social benefit programs.

Figure 1 illustrates the structure of a generalized databank system and displays the more prominent lines of communication between its elements. Note, however, that the elements of a databank system need not be unique. Multiple roles and overlap in functions are common in existing databank systems. For example, the controller, custodian, and user may be the same agency or group of persons.

The role of a subject in the databank system is to provide the "raw material" (i.e., personal information about his characteristics, background, and activities) for the databank operation. The roles of the other databank system elements are to store and process these data, and to make the data available to users for making decisions affecting a specific subject, groups of subjects, or the entire society. It is also their responsibility to protect the data against misuse, intrusion and, when appropriate, the society's claim of the "right to know."

Privacy and security

Privacy, *confidentiality*, and *security* are terms that refer to the philosophical, legal, and technical aspects of the subject's interactions with other elements of the databank system.

- *Privacy* is the right of an individual to determine for himself what personal information to share with others, as well as what information to receive from others.

Relevant questions for examining possible invasions of privacy by the data collection activities of a databank system include:[8] What personal information should be collected and stored to support the users of a specific databank system? To what extent should personal information from different sources be integrated to give a unified view of the individual? Who should be allowed to use the data and for what purposes?

- *Due process*, in the context of personal information databank systems, deals with the right of the subject to know the information stored about him in a databank system and to challenge the veracity of such information.

The relevant questions here include: Should an individual be entitled to know that information about him is being collected and stored? Should he be allowed to challenge the presence, accuracy, and completeness of this information? Westin[8] points out that answers to questions dealing with privacy and due process are *political*, not technical, to be worked out by balancing the value of civil liberties against the needs of the society.

- *Confidentiality* refers to the special status given to sensitive personal information in the databank system to minimize potential invasions of privacy. Disclosure of confidential data is restricted to users and only for purposes authorized by the controller or the subjects themselves. Confidentiality is achieved by legal and procedural means,[9,10,11] and by implementing techniques of data security.
- *Data security* refers to the protection provided to the databank system against deliberate or accidental destruction, and unauthorized access or modification, of the data. In the context of this paper, data security refers to technical and procedural means for protecting the data from intruders.

Within the databank system, the controller determines the nature of personal data to be gathered and a method of collection that satisfies the right of individuals for due process and establishes policies and procedures for data confidentiality. The collector and custodian have the responsibility to enforce the confidentiality policies and to provide procedures and technical safeguards for data security (see Figure 1).

Classification

The nature of the databank ownership, the principal use of the data, and the characteristics of the computer facilities strongly affect the complexity of the data security problem. It is useful, therefore, to establish a *classification system* that reflects data security requirements.

- *Public-Private—Public* databank systems are operated by government agencies. The controller, custodian, and users are legislative, judicial, or executive entities. *Private* databanks are operated by corporations or institutes within applicable laws. For example, the operation of credit information bureaus is regulated by the Fair Credit Reporting Act of 1970.
- *Statistical-Dossier—Statistical* databanks are operated to produce statistical summaries. Individuals are not identified in the output, but identification may be needed in the databank to permit either periodic updating of longitudinal studies or linking with other databanks. In *dossier* databanks, personal data are used to take action on specific individuals. Precise subject identification is important. Dossier databanks can be used for statistical purposes. The converse, however, is not necessarily true.
- *Centralized-Decentralized*—A *centralized* databank consists of one databank. In a *decentralized* databank, there are several physically separated databanks, each containing a part of the overall data collection. The several databanks may or may not be connected by a communication network. For example, the U.S. Internal Revenue Service maintains a decentralized databank system of income tax information.
- *Dedicated-Shared*—In a *dedicated* databank implementation, the computer facility is used exclusively to serve the databank. In a *shared* system, other databanks or computer applications use the same computer facilities.
- *Off-line-On-line*—An on-line databank permits direct real-time interaction of a user with the data through a terminal. Access may be direct or indirect. In the latter case, a databank employee acts as an intermediary. In an *off-line* databank, the user is neither in control of data processing nor knows when his data request is processed.

These classifications permit ranking databank systems in order of increasing complexity of potential data security problems, ranging from the public, statistical, centralized, dedicated, off-line databank systems (e.g., the U.S. Census Bureau), which can be expected to have relatively simple data security problems, to the private, dossier, decentralized, shared, on-line databank systems (exemplified by commercial credit bureaus and the future computer utilities), where every conceivable data security problem is likely.

Threats and countermeasures

Threats to data privacy, confidentiality, and security in a personal information databank system may arise from *all* elements of the databank system. For example, without the consent of the subjects, the controller may change disclosure rules; the custodian, collector, or users may disregard confidentiality procedures or use data for unauthorized purposes; the databank personnel, users, or even the subjects themselves may become intruders; and the databank equipment or programs may fail and cause accidental disclosures or data modification.

Technical means by which the intrusion may be perpetrated include deception, nullification, circumvention of existing protective features, and wiretapping of communication links. Whether or not the intrusion threats actually materialize depends on the nature of the data stored, the potential value of the data to the intruder, the risks he is willing to accept, and the resources he is willing to invest.

Countermeasures against the various threats include legal sanctions to deter confidentiality violations by the personnel and authorized users of the databank system, application of irreversible transformations on data in statistical databanks, and implementation of access control, threat monitoring, and cryptographic techniques.[1,2,7,12]

The design criteria for data security systems include effectiveness, economy, simplicity, and reliability. Although social policy may prefer protection of confidentiality at any cost, the rational approach to security system implementation is to protect only the data worth protecting. The following section outlines a model of the economic interactions of a rational protector of the databank system and a rational, profit-motivated intruder. This model can be used to discuss the design of cost-effective data security systems for various classes of databank systems.

A MODEL OF PROTECTOR-INTRUDER STRATEGIES

Consider the case where economic profit motivates an intruder to attempt penetration of a personal information databank system. In particular, assume that the intruder wants to compile a "mailing list," L, of N information items, each of which has the market value k. The total market value, V, of the list L is then

$$V=kN \tag{1}$$

To perpetrate the databank penetration, the intruder makes an investment, X. If the intruder requires a minimum profit, rX, $r>0$, then his maximum investment to obtain the list L is

$$X=kN/(1+r) \tag{2}$$

where it is also assumed that this intrusion is an isolated event that does not significantly benefit from previous, nor contribute to future, intrusions. The possibility of selling multiple copies of the list could be easily accommodated. The intruder's investment, X, is an expected value and should take into account the probability of failure and the risk that the databank's deterrence and retaliatory mechanisms may lead to additional costs.

To counter this intrusion threat and others, the protector of the databank system expends Y resources for data security measures. This investment should reflect the value of the protected information to the subjects, to the protector himself, and to potential intruders. Thus, prudent investment decisions of the protector would be:

- Not to commit large resources to protect information of little value to the potential intruders, even if the subjects are very strongly against the possible acquisition of this information by the intruders.
- Not to expend large resources to protect information whose release would not greatly disturb the subjects, even if the information would be valuable to the intruders.
- To commit most resources to protect information that is valuable to the intruders, and whose acquisition by the intruders would be very detrimental to the subjects.

Consider the protector-intruder interaction further. Let $I(X, Y)$ be the expected amount of information obtained by the intruder when he expends X amount of resources to overcome the Y amount invested by the protector. $I(X, Y)$ is an expected value since the probability of success for the intruder is not necessarily unity. For example, the intrusion may be thwarted because of the intruder's incomplete information about the databank's security system or even by a computer error.

As is apparent from the previous discussion of the nature of X and Y, $I(X, Y)$ is not a simple function of X and Y. However, some of its elementary properties are

- $I(O, Y)=I(X, \infty)=O$, for $X, Y>O$;
- $I(X, Y)$ is monotone non-decreasing in X and monotone non-increasing in Y.

Let $f(N)$ be the value to the intruder of N units of information and $g(N)$ be the cost to the protector and subjects of the same N units of information, occurring as a result of the intruder acquiring this information. Then, for given X and Y, the expected *net profit* of the intruder, $v(X, Y)$, is

$$v(X, Y)=f(I(X, Y))-X \tag{3}$$

while the *net loss* to the protector and subjects, $u(X, Y)$, is

$$u(X, Y)=g(I(X, Y))+Y \tag{4}$$

Given sufficient information regarding the expenditures of the protector, Y, and the nature of the security system implemented, an intruder may vary his investment, X, to maximize the expression (3). A rational protector would utilize his estimates of the value of protected information, the technical feasibility of threats, and the likely resources of the intruders to vary his expenditures, Y, to minimize the expression (4). It follows that if f, g, and I are suitably differentiable in a region containing X and Y, the selected values of X and Y will satisfy

$$f'(I(X, Y))\partial(X, Y)/\partial X=1 \tag{5}$$

$$g'(I(X, Y))\partial I(X, Y)/\partial Y=-1 \tag{6}$$

where the prime denotes differentiation.

If one or more of the functions I, f, or g are not differentiable in the region containing (X, Y), then the expressions (5) and (6) must be replaced by more complex conditions.

To use the above interaction model, analytical or empirical expressions are required for

- The value of personal information to the intruder (i.e., the function $f(N)$).

- The value of personal information in the databank to the protector (i.e., the loss function $g(N)$).
- The amount of security provided by various data security techniques (i.e., the expected expenditures, X, of intruder's resources).
- The costs of implementing the security barriers.
- The tradeoff relations between the amount of security (intruder's cost) and the protector's cost.

These items are difficult to determine and are often sensitive to the particulars of a databank security system and the information protected. There are, however, certain general features that can be discussed in qualitative terms.

VALUE OF PERSONAL INFORMATION

Securing personal information in a computerized databank system requires estimating the value of protected information to the potential intruders, the subjects of the data themselves, and the protector-custodian of the databank system. In general, this is a difficult task involving emotional as well as economic considerations. The following discussion represents only a preliminary exploration of this problem.

Value to potential intruders

A flourishing market for information has always existed. The value of trade secrets, marketing information, new product plans, and customer lists that are acquired by intruders in industrial espionage operations amounts to millions of dollars annually.[13,14]

The value of personal information to potential intruders is more difficult to estimate. A personal information market exists for mailing lists of names and addresses of persons satisfying selected criteria. These are used mainly for mailing advertising literature or making sales calls, but they are also sought for political and even criminal purposes. The mailing list rates for advertising purposes are approximately $10 per 1000 names;[15] this price increases with sophistication of selection criteria. Currently, the sale of name and address lists compiled for public information is not illegal and is practiced at all levels of government agencies. However, Federal legislation is pending[26] to make illegal such sales without the consent of the subjects involved.

The value of information on specific individuals can be expected to vary from next to nothing to thousands of dollars, depending on the prominence of the individual, the nature of the information, and his susceptibility to blackmail, political smear, or litigation.

Given the relatively high cost of penetrating the security barriers or subverting the employees of a databank system, it is likely that intrusions involving personal information are likely to be bulk operations—large numbers of information items would be obtained per intrusion, or many intrusions would be attempted to amortize the initial expenditures.

Prime-target personal information includes information held confidential by Federal or state statutes (criminal justice, public health, psychiatric, financial status, family background, etc.). Such information could be utilized for perpetrating frauds, high pressure sales, and blackmail. Illicit "purging" of records for a fee, or planting of fabricated information, may be attempted.

Court records, statistics on fraud and blackmail, and mailing list prices may provide the initial empirical data on the value of personal information to the intruders.

Value to the subject

The value to the subject of protecting his personal information can range from very little (for much of the population who, at most, would be annoyed by sales literature or salesmen's telephone calls), to thousands of dollars for those vulnerable to blackmail or character assassination. Indeed, the value to intruders of the latter type of information stems directly from the value that the subjects place on the same information, as evidenced by their willingness to pay.

The value of information of certain categories (e.g., family background) may be a time-varying function of contemporary mores. Empirical data on value of information can be gathered from statistics on the use of unlisted telephone numbers and the effects of fees for this service; the insurance premiums paid by municipalities, banks, credit bureaus, and other personal information handlers against "invasions of privacy" lawsuits; the willingness of individuals to accept money, and how much, in exchange for releasing personal information; and surveys of attitudes concerning privacy.[17]

Considerable collections of such statistics, and correlation with various population groups, are required to establish even first-order guidelines on estimating the value of personal information to individuals themselves.

Value to the protector

The value of personal information in databank systems manifests itself to the protector as:

- The legal liability of the custodian to damages

incurred by subjects whose data has been divulged to intruders through inadequate security measures or through personnel negligence. This reflects itself in the insurance premiums and payments for damages that the databank may have to make in addition to insurance coverage.
- The pressure on the custodian by the controller may result in firing of personnel, cuts in budget, restrictions of operations, etc. The dollar values of such losses could be estimated from analogous actions taken against agencies other than databanks.
- The cost of re-creating the files in cases of data destruction.

It is apparent that the functions $f(N)$ and $g(N)$, representing the value of N items of information to the intruder, the protector and the subjects, respectively, cannot quantitatively provide for all possible situations. In a more complete protector-intruder interaction model, N would be a multidimensional vector whose components represent types of information, rather than a scalar.

AMOUNT OF SECURITY AND COSTS

The *amount of security* provided by a data security technique refers to effectiveness against intrusion. As suggested previously, an intruder's expected expenditure of resources in overcoming a security barrier may be a suitable measure.

Before attempting to penetrate a databank security system, an intruder must:

- Obtain sufficient *information* about the databank system to determine whether it contains the desired information; what data security techniques are applied; what is the probability of success; and what are the penalties for failure.
- Formulate an acceptable *intrusion plan* to satisfy the cost constraints, and provide acceptable probabilities for success and risk.
- Gain *physical access* to the databank system either directly through a terminal, communication links, computer, etc., or indirectly through an employee of the databank system.
- *Penetrate* into the databank; nullify or circumvent the data security techniques to gain access to the information; acquire the information for subsequent analysis; and escape detection and reactive measures sufficiently long to complete the action.

The objectives of a security system are to deter a profit-seeking intruder by raising the intrusion cost to a level that reduces his expected profits to an unacceptable level, and to prevent access by intruders not economically motivated through effective access control and threat monitoring techniques. Effective integrity management programs must be implemented to maintain personnel loyalty and reliability of equipment and software.

These three classes of data security techniques must be applied against intruders to:

- *Deny information* about the security system. It may not be possible, or even not desirable,[18] to maintain secrecy about the security techniques used, but the specific access codes and keys must be kept from all but a few authorized personnel.
- *Prevent unauthorized access* to the computer system (terminals, communication links, processor, data storage devices), the protected data files within the computer, and to specific data processing operations.
- *Detect intrusion attempts*; discriminate among threats; sound alarm; and take responsive action.
- *Maintain integrity* of the databank system by reducing opportunities for personnel subversion, increasing hardware and software reliability, and controlling any changes in software or hardware.

The amount of security

The burden of *preventing* intrusion is borne by the access control techniques. Threat monitoring is used mainly to reduce the *time* available for perpetrating the intrusion and for *post facto* investigation.

The basic elements of access control are:

- *Authorization* of persons to access the computer facility, terminals, data files, and processing operation.
- *Identification* of a person seeking access.
- *Authentication* of his identity and access authorization.

Not all databanks have implemented all of the above steps as part of the access procedure—in some, the mere possession of a valid password is considered sufficient.

The *enforcement* of access control techniques may be assigned to computer facility personnel, performed by hardware devices, or implemented in software.

To *defeat* an access control technique, an intruder must be able to accomplish one of the following:

- *Acquire or forge* the proper identification and authentication passwords or keys.
- *Curcumvent or disable* the access control technique.

The choice depends on the technical feasibility of these approaches and, for those deemed feasible, the relative costs, risks, and required time.

Acquisition of access control information

The protective capability of passwords and privacy transformation keys lies in the intruder's uncertainty regarding which of the very large number of possible passwords or keys is being used. For example, there are $26^5 \approx 1.2 \times 10^7$ possible 5-character and $26^6 \approx 3.1 \times 10^8$ possible 6-character passwords.

Nevertheless, a trial and error search for the correct password is not entirely infeasible: a minicomputer can be programmed to imitate the databank terminal's sign-on and password sending sequences. This computer can then be used to try different passwords at the rate permitted by the communication channel and the databank computer. The intruder's effort is greatly reduced if the passwords used by the databank are selected for their mnemonic capability (i.e., are similar to English words). For example, studies of 5-character alphabetic code words that were required to differ in at least two characters and contain at least two vowels show[19] that only 150,480 5-character words can be selected out of the total space of 1.2×10^7. To test all of these at the rate of 10 per second would require slightly more than 4 hours.

However, passwords could be obtained with less effort by wiretapping the communication links and recording the sign-on sequences.[20] Acquisition by wiretapping of passwords that are used once-only requires more sophisticated techniques, e.g., "piggy backing":[1] insertion of a minicomputer in the line to intercept user-computer communications, to return an error code to the user, and to enter the file with the password obtained. If passwords are generated by a pseudorandom process for once-only use, and several passwords are intercepted, certain number-theoretic techniques may be applied to discover the password generation process and its parameters.

The intruder's cost of acquiring passwords through wiretapping ranges from the cost of recording equipment—a few hundred dollars, to the cost of a minicomputer and associated programming—a few thousand dollars. The risks include the possible legal prosecution.

Cryptanalysis of privacy transformations

The intruder's work factor in attempting to solve for the key of a *privacy transformation* from intercepted, enciphered data is normally much larger than required for passwords. The key spaces are much greater, and exhaustive trial-and-error solution is infeasible. However, analysis of intercepted transformed data from the point of view of language *statistics* can be applied. Relevant are

- Single character frequency distribution;
- Digram (pairs of characters) and polygram frequency distributions;
- Word usage patterns;
- Syntactical rules of the language.

The two main classes of privacy transformations are *substitutions* of characters in the data with other characters (or groups of characters) and *transportation* of the order of the characters.[21,22,23]

The easiest to apply in a computer system are the substitution transformations:

- *Monoalphabetic substitution*, or the "Caesar cipher," where each character, x_i, of the data (the "plaintext") is transformed into a character, y_i, of the "ciphertext" by modulo N addition of a constant c

  $$y_i = x_i + c \qquad (\text{mod } N)$$

  where N is the size of the alphabet. The constant c has only $N-1$ possible values and, thus, can be easily discovered.
- *Polyalphabetic substitution* of period u (the Vigenére cipher) consists of cyclic application of u monoalphabetic substitutions by adding modulo N the constants $c_0, c_1, \ldots, c_{u-1}$ so that

  $$\left.\begin{aligned} y_0 &= x_0 + c_0 \\ y_1 &= x_1 + c_1 \\ &\cdots \\ y_j &= x_j + c_{j(\text{mod } u)} \end{aligned}\right\} \quad (\text{mod } N)$$

  The key space here contains N^u possible selections of the constants $c_0, \ldots, c_{u-1}$.
- *A k-loop polyalphabetic substitution* uses k sets of alphabets, applied cyclically with periods $u_1, \ldots, u_k$:

  $$y_j = x_j + c_{1,,j(\text{mod } u_1)} + \cdots + c_{k,j(\text{mod } u_k)} \qquad (\text{mod } N)$$

  where the u_i are relatively prime (mod N)
- A *Vernam cipher* is a polyalphabetic substitution (Vigenère) where the key period is at least as long as the amount of data to be transformed.

Computer-aided solution of substitution transformations has been studied by Tuckerman.[24] Such solutions can always be found, provided that sufficient contiguous lengths of transformed data (ciphertext) can be acquired). If the ciphertext contains fragments of known data, even if their precise location is not known, the cryptanalysis task is greatly simplified. In the case of highly formatted artificial languages (programs), where the fixed vocabulary is very small and used with rigid observance of syntax and punctuation rules, fragments of known plaintext are very likely. If the polyalphabetic cipher keys are relatively short and coherent (phrases of a natural language), the task is even further simplified.

The techniques for solving substitution-type transformations proceed as follows:[24]

- A *Caesar cipher*, where the key consists of a single constant, is easily solved by language statistics or trial and error. Shannon[21] has shown that for natural language plaintext in English, the sufficient length of a fragment of intercepted ciphertext (the unicity distance) is about 30 characters.
- A 1-*loop polyalphabetic* (Vigenѐre) cipher of period u is reduced to u Caesar cases by statistical analyses and trial-and-error determination of the key period, u. At least 20 u characteris of intercepted text are required.
- A 2-*loop polyalphabetic* cipher of periods u is reduced to one loop case by certain "differencing" methods.[24] Then, the 1-loop analysis can reduce the problem to Caesar cipher level. At least $100(u+v)$ characters of ciphertext are required. The effort is considerably greater than for the 1-loop case.
- The Vernam cipher (where the key is as long as the data, used only once, and generated by a natural random process) cannot be solved. However, if the key is generated by a pseudo-random process, such as a shift-register sequence generator, and plaintext fragments are known, then computer-aided trial-and-error methods may lead to a solution.

The intruder's work factor in the above cryptanalytic activities requires a sufficiently powerful computer and appropriate cryptanalytic programs. Given these, solutions are sometimes found in minutes.[24] To successfully attack privacy transformed data requires an investment measured in thousands of dollars for the more complex systems. The work factor is in terms of hundreds of dollars if simple substitutions are used.

Circumventing or disabling of access controls

Circumvention of access controls enforced by databank *personnel* can be attempted by using the well-developed techniques of diversion, confusion, or intimidation. Costs are low and risks involve being "kicked out," which in turn might be good diversion for permitting an accomplice to enter. Personnel other than professional security guards are well-known for their reluctance to challenge others not known to them.

Hardware access control devices (e.g., locks operated by keys or controlled by programs) are usually effective, especially if connected to alarm systems.[25] However, some types could be easily disabled, thus reducing the enforcement to facility personnel. Assistance of unsuspecting facility employees could be recruited with the "forgot my key" gambit. Costs and risks are low.

Circumvention of *software* enforced access controls (i.e., the protective features of operating systems) requires that the intruder gain not only access to the computer through regular or illicit terminals, but also the ability to enter programs into the system. Diversion and "flooding" techniques may be able to overwhelm the threat monitoring system long enough to perpetrate the intrusion.[6] The resources required by the intruder include a computer to develop and test the intrusion plan and programs. The risk is low. However, the operating systems designed for high security[3,4] may escalate the intrusion costs into the thousands or even ten thousands of dollars.

Protection costs

The costs involved in implementing a data security system include the initial planning and design, initial investment in hardware devices and software, the recurring operating costs, and the decreases in functional capability. The available cost data is very limited and does not suffice for formulating analytic expressions for the protector-intruder interaction mo-el described above.

Hardware access devices, such as card-key locks for doors or computer terminals, are priced in the $150-300 range per unit. Complete systems start from $5000. Hardware implemented data privacy systems for communication links cost in the $2000 range per unit.

Data on software implementation of access controls in operating systems is equally scarce. The following represents almost the entire cost data base:[3,5]

Main memory requirements:	10-20%
Programming time:	5%
Operating system code:	10%
Recurrent CPU time:	5-10%

Some cost data points are also available for the implementation of privacy transformations in software.

In substitution type privacy transformations, each character of plaintext is transformed into a character of the ciphertext by addition of one or more constants, c_j. Also required are similar decoding and the necessary key-retrieval operations. In terms of the percent of the databank operating system overhead, the following computing time requirements have been established for applying privacy transformations to 10-bit characters in a CDC 6600 computer:[7]

One-time Vernam ciphering:	0.66%
Vigenère ciphering (table look-up)	3.5%
Vigenère (modulo arithmetic)	6.3%

The above cost figures are quite sensitive to the type of information retrieval system used and represent only isolated cost data points. Estimates of decreased functional capability of the databank system caused by security requirements are even less available. A systematic effort to compile a comprehensive data base of security system costs and decreases in functional capability is clearly needed.

CONCLUDING REMARKS

The design of cost-effective data security safeguards for personal information databank systems requires a careful balancing of the value of protected information against the protection costs. In particular, it is important to consider not only the value of personal information to the subjects, but also to the potential intruders, i.e., the protection investments should be made on a rational basis.

The simple protector-intruder interaction model discussed in this paper illuminates the nature of the protector's investment problems when faced with an equally rational intruder. However, before this or any other interaction model can be fully utilized, it is necessary to formulate appropriate analytical or empirical relationships among the value of information to the parties involved, the costs of protection and intrusion, and the effectiveness of data security and intrusion techniques. Deriving such relationships and gathering empirical data will be a major objective of the authors' further work in this area.

ACKNOWLEDGMENTS

The authors would like to acknowledge valuable suggestions and comments by their colleagues at The Rand Corporation, Mario L. Juncosa, Irving S. Reed, and Selmer M. Johnson, and by Robert H. Courtney of the IBM Corporation.

REFERENCES

1 H E PETERSEN R TURN
System implications of information privacy
AFIPS Conference Proceedings 1967 SJCC Vol 30
pp 291-300

2 W F BROWN
AMR's guide to computer and software security
AMR International Inc New York 1971

3 C WEISSMAN
Security controls in the ADEPT-50 time-sharing system
AFIPS Conference Proceedings 1969 FJCC Vol 35
pp 119-133

4 G S GRAHAM P J DENNING
Protection—principles and practice
AFIPS Conference Proceedings 1972 SJCC Vol 40
pp 417-429

5 C WEISSMAN
Trade-off considerations in security system design
Data Management April 1972 pp 14-19

6 D VAN TASSEL
Computer security management
Prentice-Hall Inc Englewood Cliffs New Jersey 1972

7 W A GARRISON C V RAMAMOORTHY
Privacy and security in databanks
Technical Memorandum No 24 Electronics Research
Center University of Texas Austin Texas November 2 1970

8 A F WESTIN
Civil liberties and computerized data systems
in M Greenberger (Ed) Computers, Communications
and Public Interest Johns Hopkins Press 1971

9 A F WESTIN
Privacy and freedom
Atheneum New York 1967

10 A R MILLER
Assault on privacy: computer databanks and dossiers
University of Michigan Press Ann Arbor Michigan 1971

11 P NEJELSKY L M LERMAN
A research-subject testimonial privilege: what to do before the subpoena arrives
Wisconsin Law Review Vol 1971 No 4 pp 1085-1148

12 R TURN H E PETERSEN
Security of computerized information systems
Proceedings Carnahan Conference on Electronic Crime
Countermeasures University of Kentucky Lexington
Kentucky 1970 pp 82-88

13 R DONOVAN
Trade secrets
Security World April 1967 pp 12-18

14 P HICKSOM
Industrial espionage
Spectators Publications Ltd London 1968

15 *Firms sue in mailing list theft*
Computerworld 8 July 1970

16 *Security breach leads to police data theft*
Computerworld 10 February 1971

17 *A national survey of the public's attitudes toward computers*
AFIPS-Time Inc New York & Montvale New Jersey
November 1971

18 P BARAN
On distributed communications: IX, Security, secrecy and tamper-free considerations
The Rand Corporation RM-3765-PR August 1964

19 W F FRIEDMAN C J MENDELSOHN
Notes on code words
American Mathematical Monthly August 1932 pp 394-409

20 J M CARROLL
The third listener
Dutton 1969

21 C E SHANNON
Communication theory of secrecy systems
Bell System Technical Journal 1949 pp 656-715

22 D KAHN
The codebreakers
The Macmillan Co New York 1967

23 M B GRIDANSKY
Cryptology, the computer and data privacy
Computers and Automation April 1972 pp 12-19

24 B TUCKERMAN
A study of the Vigenére-Vernam single and multiple loop enciphering systems
IBM Corporation Report RC 2879 14 May 1970

25 R J HEALY
Design for security
John Wiley & Sons Inc New York 1968

26 *U.S. Senate Bill S.969*
U S Senate 25 February 1971

Snapshot 1971—How Canada organizes information about people

by JOHN M. CARROLL

University of Western Ontario
London, Canada

INTRODUCTION

In 1971 the Government of Canada initiated a study to determine whether the computerization of personally identifiable records concerning or describing Canadian residents would diminish their quality of life or adversely affect their life chances, and to propose remedial action in the event this premise proved to be true.

The study was carried out by a joint Task Force appointed by the Departments of Communications and Justice. The empirical studies group of the Task Force was charged with determining the magnitude and composition of personal data banks in the public and private sectors and the means by which such data are gathered, processed, stored, and disseminated. This paper summarizes the results obtained by this group.

The investigative procedures used consisted of soliciting briefs from organizations thought to be interested in the subject, making formal site visits to selected firms and agencies, conducting field studies to gather background information on the organizations to be visited, sending letters of inquiry to multinational organizations, and mailing a detailed questionnaire to all Canadian organizations believed to possess significantly large files of personal data that were or might become computerized.

Although the site-visit technique provided the principal input regarding large government data banks such as those of the Royal Canadian Mounted Police, Statistics Canada, the Department of National Revenue/Taxation, and the Department of National Health and Welfare, it was the questionnaire which provided the most comprehensive information of a quantitative nature.

Over 2,500 questionnaires were mailed and the response exceeded 50 percent. What emerges from this portion of the study is a finely detailed snapshot of how one developed nation makes use of information handling technology in the management of personalized information.

In the largest sense, the most significant thing about this study is that in Canada concern regarding potential invasions of individual privacy of information abetted by computers arose initially within the federal government. This fact is borne out by a tabulation of responding organizations who indicated that they had received complaints from the public regarding their data handling practices. Only 16 percent reported receiving any.

Nature of Complaint	*Number of Respondents*
Inadequate provisions to review one's own record	200
Methods of collecting personal information	180
Practices of disseminating personal information	160

CHARACTERISTICS OF THE RESPONSE BASE

Organizations which replied to the questionnaire employed about one-sixth of the labor force. Thus the questionnaire returns, with due allowance for potential distortions, represent a comprehensive overview of Canadian data banks—or, more specifically, of data banks containing identifiable personal data about individuals. The largest number of respondents employed less than 100 persons each: 23 percent of the respondents had 80 percent of all employees.

	Average	*Total*
Employees	980	1,200,000
Customers	61,000	65,000,000
Subjects	70,000	24,000,000
Data Recipients	4,900	2,000,000

"Customers" were defined as including present clients, customers, patients, students, policy-holders, and members (of associations). Many Canadians are customers of several organizations. The most numerous group had between 2,000 and 25,000 customers each; 14 percent of the respondents had 83 percent of all customers.

Only 40 percent of the respondents said they had files on individuals regarded by them as "subjects", defined to include prospective customers, persons upon whom credit or criminal records are held, auto registrants, and subjects of research studies. Federal agencies dealing with veterans affairs, family allowances, and manpower and immigration responded under this denomination as did some provincial public health agencies. The most numerous group had less than 1,000 subjects each; 7 percent of the respondents had 51 percent of all subjects.

With regard to information recipients, only 37 percent of respondents admitted to having any; 16 percent of the respondents served 95 percent of information recipients.

Thus, there comes into focus the picture of an information elite that uses vast files of personalized information as its base of power.

CHARACTERISTICS OF FILES

The files reported upon contained over 83 million records. Respondents in the most numerous classification had fewer than 5,000 records each; 19 percent of the organizations held 90 percent of the records. It was our practice to request information on what we perceived to be the largest file held by a particular questionnaire recipient.

Average Characteristics of Files

Size of file	72,000	records
Size of record	520	characters
Number of requests for information	1,300	per year
Period of retention (inactive records)	67	months

With regard to size of record, the largest response category had record sizes under 300 characters. This was offset by 90 organizations whose record sizes exceeded 2,000 characters.

Over a million requests for information were reported on a yearly basis; 791 respondents said they had fewer than 100 requests a year, while 46 organizations said they answered more than 10,000 requests a year.

Organizations which reported that they responded to more than 10,000 requests for information annually regarding persons in the "subject" category included credit bureaus, police forces, motor vehicle bureaus, and mailing-list suppliers.

With regard to the time a record is held after an individual has severed his connection with the organization, 534 respondents said they keep such records seven years or more.

COLLECTION OF DATA

The subject himself is the prime source of information. Health services are in second place. One would expect references to be checked, but it is interesting that they turn out to be more important sources than former employers, present employer, or educational institutions. We find it significant that published records are rarely consulted and that law enforcement agencies are sources at all. Figure 1 shows the relative utilization of the more common sources of information; Figure 2 shows the relative use of less common sources.

We found that the data gatherers most likely to tap medical sources included health services, insurance companies, social welfare agencies, charitable institutions, and regulatory agencies. Data gatherers most likely to approach present or former employers included merchandising houses, employment agencies, insurance

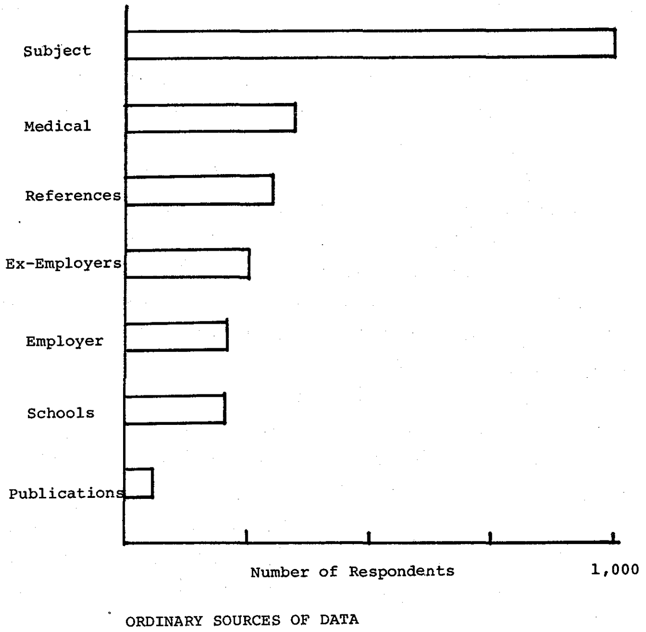


Figure 1—Commonly used sources of information concerning or describing individuals

companies, police forces, and prospective employers. Agencies most likely to interview a subject's family included health services, social welfare agencies, charitable institutions, and police forces. Organizations most likely to interview a subject's neighbors included health services, educational institutions, insurance companies (through credit bureau representatives), police, and social welfare agencies. Police forces reported that they principally consulted other police, regulatory agencies, private investigators, insurance companies, and employers. Private investigators reported that they obtained information from police, insurance companies, other private investigators, social welfare agencies, and regulatory agencies.

Among the techniques employed by data gatherers, protection of informants outranked confirmation of facts from independent sources in importance to the data gatherer.

In response to the questions as to whether the individuals upon whom records were kept or groups representing their interests ever complain against the method of collecting any item of information, five organizations said they get frequent complaints, 910 said they get none at all.

Most likely to receive complaints regarding methods of collecting personal data are law-enforcement agencies, motor vehicle bureaus, credit bureaus, travel-and-entertainment card companies, and insurance companies.

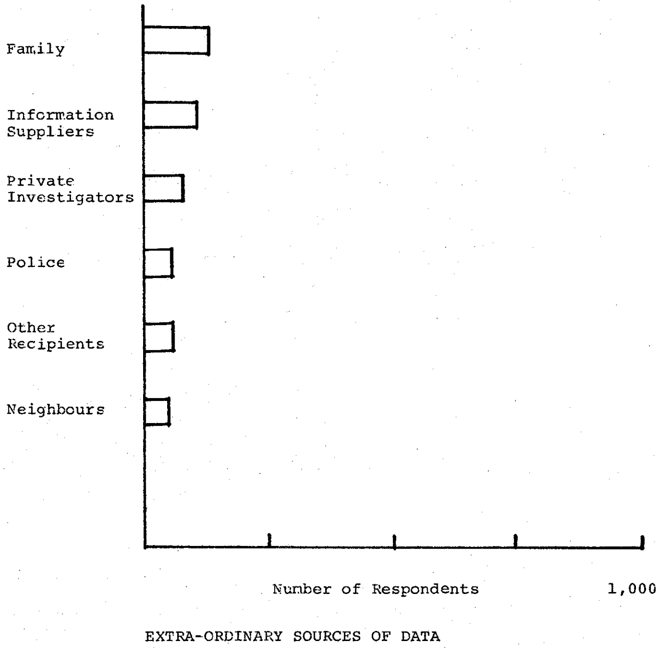


Figure 2—Less commonly used sources of information regarding individuals

CUSTODY OF INFORMATION

As to management policies regarding disclosure of personal data, 55 percent of respondents said they have an unwritten policy, 33 percent have a written policy, and the rest have none at all. Non-profit institutions were twice as likely to have a written policy than were profit-making organizations.

We inquired whether an explicit statement of the organization's policy was communicated. Responses revealed that it is highly likely that, where such a policy exists, it will be communicated to employees charged with records management but unlikely that it will be communicated to either the subjects of the records or to the general public.

As to policing the actions of staff with regard to misuse of personal information: 23 percent of respondents do not police the actions of their own staff; 67 percent do police the actions of staff but claim they don't catch any offenders; 10 percent police the actions of staff, catch some offenders, and prosecute or discipline the ones they catch.

With respect to the likelihood that an organization will take effective action against its own employees for misuse of personalized information in its files, non-profit institutions were nearly twice as likely to take effective action than were profit-making organizations.

The organizations most likely to take effective action were motor-vehicle bureaus, police, public utilities, credit bureaus, and health services.

Response to the question as to whether individuals on whom records are kept or groups representing their interest ever complain about disclosure of personal information revealed that four organizations get frequent complaints; 873 get none at all.

Most likely to receive complaints regarding disclosure of personal data were motor vehicle bureaus, credit bureaus, educational institutions, law-enforcement agencies, social welfare agencies, and employment agencies.

DISSEMINATION OF INFORMATION

Regarding exchange of information with other organizations 38 percent of respondents said they did exchange information; 62 percent said they did not.

Most likely to disclose personal data outside their own organizations are motor vehicle bureaus, regula-

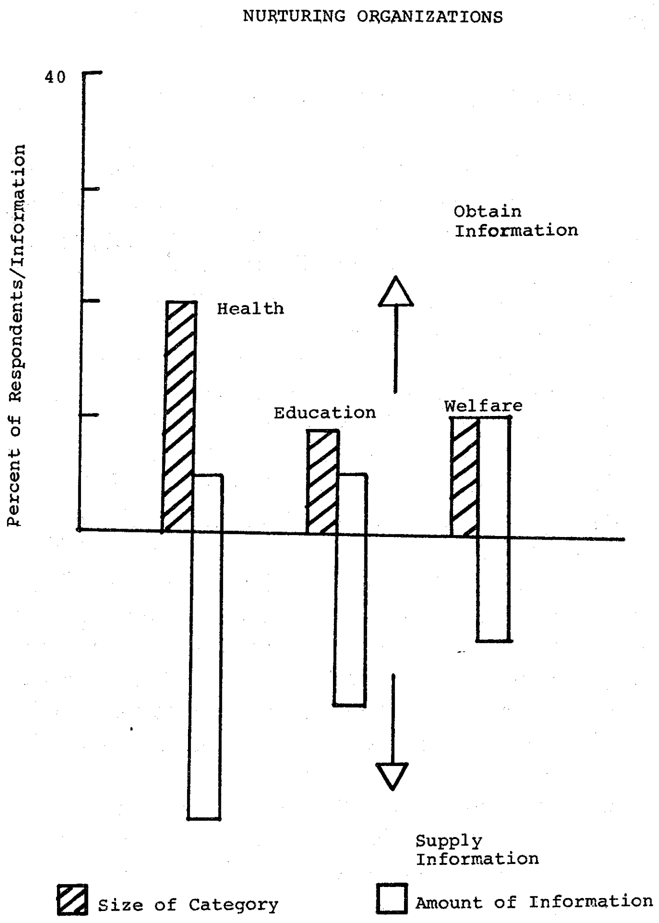


Figure 3—Information interchange patterns of nurturing or subject-serving organizations show them as sources

tory agencies, educational institutions, credit bureaus, health services, insurance companies, oil companies, and law-enforcement agencies.

Information is most commonly furnished in response to specific requests. Publication of periodic reports for widespread distribution is a rarity.

EXCHANGE OF INFORMATION

We utilized information developed by analysis of responses to the Task Force questionnaire to construct a matrix illustrating the degree of exchange of personalized information among organizations.

We found it convenient to classify these organizations as nurturing, that is concerned principally with the well-being of the individual; business, that is dealing with the individual on a quid-pro-quo basis; and authoritarian, or interested primarily in ensuring that their subjects conform to the norms of society. Nurturing organizations tend to supply information to groups in the other two categories. Business-type organizations tend to exchange information freely, principally with organizations of the same general type. Authoritarian organizations appeared to gather personal information in a volume disproportionate to their relative number in the response base and to communicate little information to other organizations. These patterns of information interchange are depicted in Figures 3, 4, and 5.

With regard to international traffic in personal information, 61 organizations said they frequently supply information to U.S. organizations; 107 organizations said they frequently obtain information from U.S. organizations.

We found only five organizations had their files entirely in the U.S.A. Organizations most likely to have

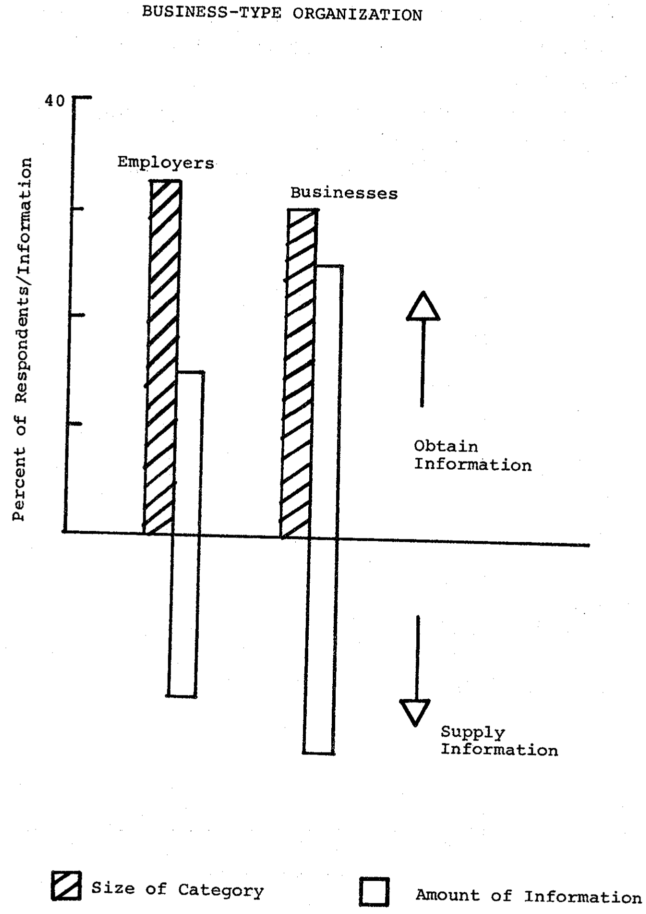


Figure 4—Information interchange patterns of business-type or self-serving organizations show them as dynamic storage elements

some files containing personal data located in the U.S.A. were oil companies, associations (especially labor unions), insurance companies, health services, manufacturers, and lending institutions. About 10 percent of all organizations employing 500 or more persons had some of their files in the U. S. A.

Ten organizations had their customers entirely in the U.S.A. and 10 organizations had their information recipients entirely in the U.S.A.

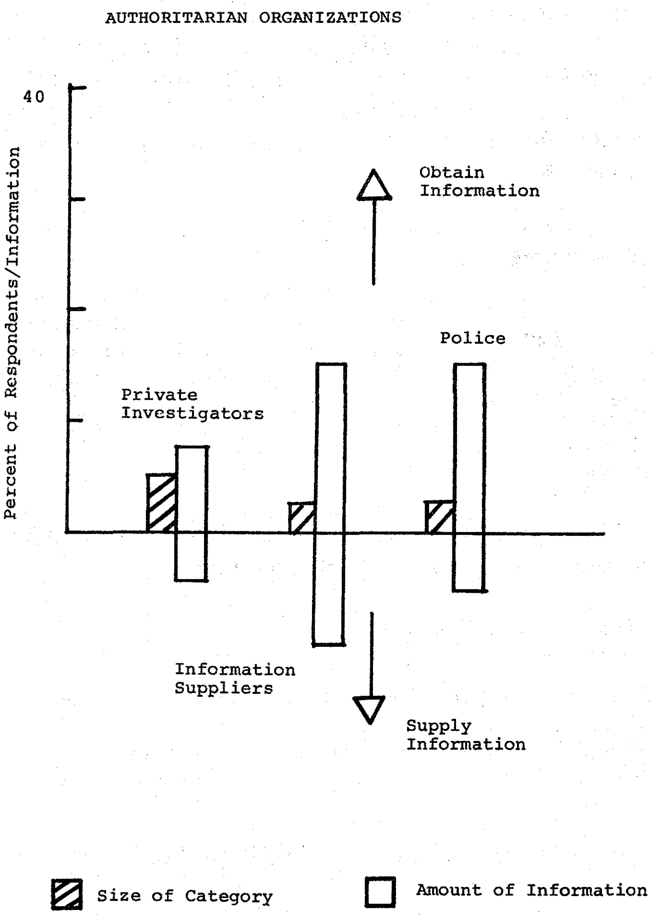


Figure 5—Information interchange patterns of authoritarian or society-serving organizations show them as sinks

With regard to future intentions to locate files in the U.S.A., more than three quarters of responding organizations said they would not do so; 57 organizations said they already had files in the U.S.A. The remainder said they would do so to save money or if they would be placed at a severe disadvantage by not doing so. Figure 6 summarizes information developed regarding exchange of data between Canadian and U.S. organizations.

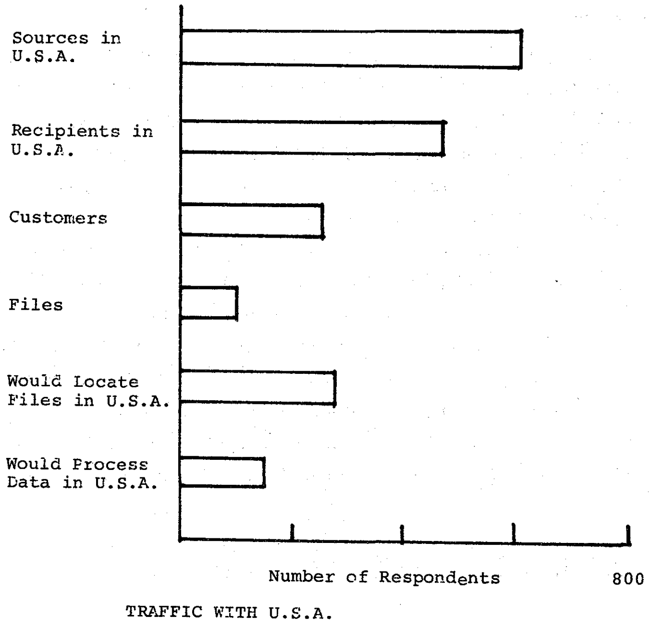


Figure 6—International information interchange: Canada-U.S. traffic

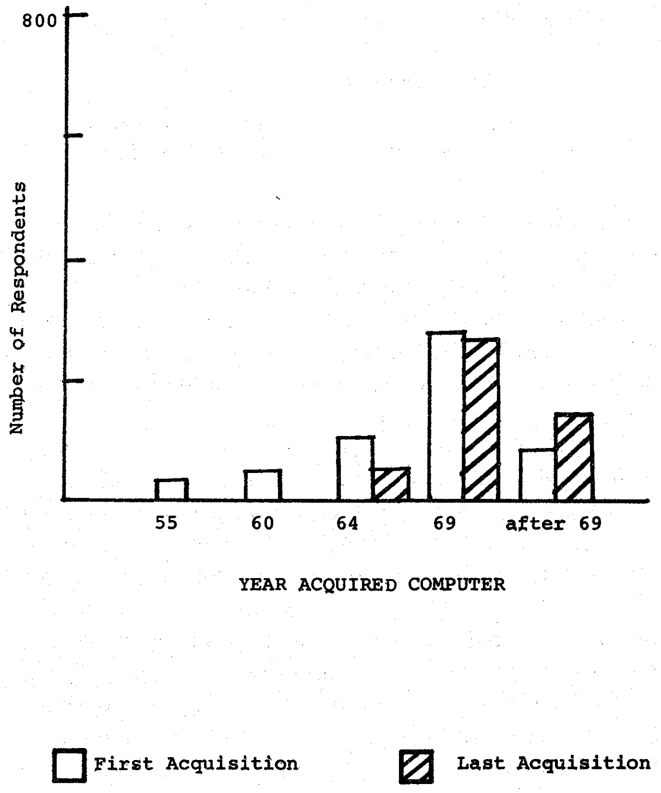


Figure 7—Trends in acquisition of central processors in Canada

EXTENT OF COMPUTERIZATION

Roughly half of our respondents (about 500) utilize electronic data processing equipment. Of these, about 300 have their own computers and 200 employ the facilities of computer service bureaus. Of the respondents having computers, about 1/3 have facilities for remote access from terminals.

The average computer user among our respondents first began computer processing of records in 1964-65 He procured his present machine in 1967; so we are looking at a group of computer users who were initiated on second generation computers and later upgraded to third generation machines, in other words, a population of sophisticated users. Figure 7 illustrates the trend in acquisition of computing equipment.

CHARACTERISTICS OF MACHINES

The average computer reported upon may be regarded as a large machine; 123 organizations have computers whose memory size exceeds 256,000 words of core storage.

Average on-line disk storage capability appears to be adequate for remotely accessed time-sharing should the user so desire.

Average Characteristics of Computers

Core Memory	133,000 WORDS
On-line disk memory	130,000,000 BYTES

One hundred twenty-four respondents said they had high-speed remote terminals. Of these, 102 had less than six terminals; 22 had six or more.

Use of keyboarded remote terminals was reported by 134 respondents; 101 had less than 12 such terminals; 26 had from 12 to 200 terminals; seven had more than 200.

Seventy-three percent of respondents have implemented physical access controls over electronic data processing equipment; 39 percent have implemented hardware or software security measures such as passwords, terminal identification codes, or cryptographic coding; 42 percent routinely seek to establish the personal integrity of processing personnel; 58 percent report utilizing audit logs or other access-monitoring methods; 69 percent employ secure disposal methods for unwanted tapes or printouts; and 31 respondents report implementing security measures beyond access control, integrity checks on processing personnel, audit logs, and secure disposal methods.

UTILIZATION OF COMPUTERS

Despite the widespread use of computers by organizations, the penetration of computers within organizations is not all that great.

Relatively few computer users report that they hold computerized records on all or most persons in any given category. Still fewer users report that they have computerized all or most information held on such persons.

The following table summarizes Task Force findings with respect to the classification of files reported upon, the percentage of respondents who hold computerized records on all or most persons in each category, and the percentage of respondents who say they have computerized all or most of the information they hold on each of these persons.

Extent of Computerization
(Percent of respondents)

Category	Subject matter of file	Hold computer records on all or most persons	Have computerized all or most information
Employees	31	58	30
Customers	55	72	40
Subjects	14	30	22

Most computer users said they supplement their machine sensible files with manual files.

Characteristics of Manual Files
(Percent of respondents)

Supplement computer files with manual files	90
Manual files contain more subjective information	83
Manual files contain more sensitive or confidential data	75
Manual files contain more narrative or graphical data	70

ASSESSMENT OF COMPUTERIZATION

The following table summarizes the assessment of computerization by organizations providing responsive answers to questions in this category (i.e., organizations using computers):

Comments Regarding Computers
(Percent of Respondents)

Detected errors in records during computerization	74
Computer improves routine data handling	51
Computer provides more complete and timely reports	45
Computer is essential to operations	41
Computer permits collation of data regarding individuals	32
Improved management planning is principal benefit of computerization	4

In addition, the importance of accuracy problems experienced with the computer was reported to be insignificant. Only 16 percent of respondents say that, as a result of increased retrieval capability after computerization, they are called upon to furnish more individually identifiable information to government agencies; and only 34 percent say that, as a result of computerization, they are called upon to furnish more statistical (aggregated) information regarding individuals.

The amount of data collected per given individual after computerization was reported to have increased. However, only 39 percent of respondents attributed this increase to the fact of computerization; on the other hand, 60 percent attributed the increased data collection to changes in organizational objectives or programs, or to increasing government requirements for collecting or reporting information.

RIGHTS OF SUBJECTS

The right of an individual to examine his own record or a copy of his record from the file is the cornerstone of many suggested reforms in the area of privacy of individual information.

Following is a complete tabulation of answers to the question of whether or not this right exists:

	Number	*Percent*
No response	64	5.27
The individual does not know the record exists	62	5.19
He has no understanding of the contents of his record	135	11.03
He can examine *all* data in his record	502	41.87
He can examine *some* data in his record	291	23.21
He can examine *no* data in his record	172	14.24

Right to Examine One's Personal Record by Type of File

	Employees	*Customers*	*Subjects*
Does not know record exists	9	43	10
Has no understanding of contents	31	85	19
All	169	257	76
Some	133	121	37
None	22	131	19

In cases where an individual is permitted to examine data in his record, we asked whether translation or interpretation was provided in an official language that the individual understands; 68 percent said it was.

Organizations least likely to permit an individual to examine all data in his record include travel-and-entertainment card companies, market research firms, insurance companies, social welfare agencies, police forces, health services, employment agencies, and oil companies.

Response to the question of whether individuals on whom records are kept or groups representing their interests have ever sought to examine their own records or complained about the adequacy of an organization's practices regarding an individual's right to examine his own record revealed that eight organizations get frequent complaints; 867 get none at all.

Most likely to receive complaints about the inability of a subject to examine his record are law-enforcement agencies, credit bureaus, and health services.

CONCLUSIONS

The acquisition of computing equipment has declined in recent years, which could indicate that the majority of those organizations who feel they could benefit from a computer have already acquired at least the main frame. However, utilization of computers for handling personal records is relatively low both in number of persons whose records are computerized (breadth) and the amount of information regarding each person who is computerized (depth). The fact that customer records tend to be most completely computerized both in breadth and depth demonstrates that the controlling factor behind decisions to establish or augment computer-based personal data banks may be based upon the expected economic return from this exercise. Thus economics rather than either technical infeasibility or unavailability of data has thus far inhibited the wholesale creation of personal data banks.

Much greater capability for remote-access computing exists than is currently being utilized. However, there is a growth trend in this area. This may have unfortunate consequences with regard to data security. With a few notable exceptions, computer users have not yet proved fully capable to safeguard the confidentiality of computer-based files that are processed in the batch mode at a central location; and remote-access computing presents a whole new dimension of hazard in respect of unauthorized interception and intrusion.

A great deal more exchange of personal information takes place than is generally appreciated. There appears to be a flow of information that proceeds through stages from nurturing organizations such as schools and health services to authoritarian organizations. Therefore, it is quite likely that personal information volunteered by an individual seeking some social benefit in one context may be used in another context to impose sanctions upon him for failure to conform to some societal norm.

International traffic in personal data by large multinational organizations is already significant in volume and may easily double in the near future. Such traffic may adversely affect the quality of life and life chances of citizens in ways which are beyond the power of national governments to ameliorate.

The official report of the Task Force is available from Information Canada under the title: "Privacy and Computers Task Force Report."

Details of empirical studies (Studies 2, 3, and 4) are available from the Department of Communications under the title: "Personal Records: Procedures, Practices, and Problems". This document contains a copy of the Task Force questionnaire and a tabulation of responses.

The reader is urged to consult also the report of Professor Alan Westin's study of the records problem in the U.S. This study was sponsored by the National Academy of Sciences. The report of a British study group was published in July 1972.

Hardware/software trade-offs—Reasons and directions

by RICHARD L. MANDELL

Compata, Incorporated
Tarzana, California

A hardware/software trade-off is the establishment of the division of responsibility for performing system functions between the software, firmware and hardware. This is part and parcel of the fundamental process of defining computer architecture. It begins the day a computer is conceived and may be carried on by an ever widening group of individuals until the last computer of a given model is retired. There are areas of the trade-off which are the sole preserve of the manufacturer and his hardware/software team. Other areas of the trade-off are the responsibility of the user, or independent equipment manufacturers.

TYPES OF TRADE-OFFS

Since hardware/software trade-offs occur in all areas of computer design and application, it is difficult to write about them without discussing most of the factors that enter into both hardware and software design. In this paper, an attempt will be made to define several classes of trade-offs and discuss the reason for each.

Some computers are microprogrammed. In these systems the microprogram resides in a fast control store and controls the flow of data through storage, transformation units and data paths. For the purposes of this paper, the microprogram will be referred to as firmware, and the control store and other functional units will be called hardware. In the discussion that follows, a conventionally organized wired logic control will be viewed as a part of the hardware.

Trade from software to hardware

The first class of trade-off is the trade from hardware to software or vice versa. Such a trade-off may involve transferring whole functions, such as memory protection from one system to the other. On the other hand, a trade may mean merely shifting the boundary between system hardware and system software by providing different instructions or architectural features.

Trade from software to firmware

Many modern computers are microprogrammed. This introduces another trade-off possibility. Rather than introduce new hardware in place of software, the trade is often made between software and firmware. This trade may sometimes have no effect other than to speed up a system by eliminating main memory fetch cycles. On the other hand since the microprogrammer has available to him data paths and parallelisms that are not available at the traditional software level, it is possible to perform functions that would not be feasible or efficient in software.

Trade from firmware to hardware

In designing a microprogrammed machine, a designer must decide which functions are to be performed strictly under hardware control and which functions are to be performed by sequences of microinstructions. He must also decide what fundamental data paths and functional units will exist within the machine. Both of these types of trade-offs constitute trade-offs between hardware and firmware. The hardware/firmware trade may be made without influencing the external architecture of the system.

Direction of the trade

Frequently, computer designers and users think only in terms of making machines bigger and faster. However, there is always a market for smaller and simpler machines as the manufacturers of minicomputers, smart terminals and desk calculators have discovered. Thus, frequently the design objective is to simplify the computer. Accordingly, we will arbitrarily consider the three elements hardware, firmware and software to form a hierarchy, with hardware at the bottom, with firmware next and with software at the top. An upward trade will then be defined as a trade in which responsibility for a function is moved through the hierarchy from hardware toward software. A downward trade moves in the other direction.

An inward/outward trade

In the list of trades considered so far, the hardware has been considered to be a single system. In many systems the hardware is really viewed as an interconnection of subsystems which may themselves be hierarchically organized (i.e., memory systems and I/O systems). The organization and function resident within or attachable to these subsystems is frequently a part of the hardware/software trade. For example, as more autonomous control is given to the I/O system, the requirement for software control of the I/O system may be simplified. Trade-offs which move function from the CPU to autonomous control units will be termed outward trades and trades in the other direction will be termed inward trades. It should be noted that the outward trade may go so far as to remove a function from the computing system completely and place it in another communicating system. This is the case when printing is removed from the main I/O system and transferred to an autonomous off-line printer. Another example of this is an architecture which allows peripherals to communicate with one another without requiring service from the software.[1]

The outward trade is an impressive tool for system enhancement after the system architecture has been frozen. This is possible because the I/O system usually presents a clear stable interface to the outside world. Thus, autonomous processors such as sorters,[2] communications handlers,[3] array processors,[4] and support processors have been attached to CPUs in order to perform functions that would otherwise be done by central processor software. Some architectures have made the I/O systems sufficiently powerful to take on the role of much of the supervisor.[5] Though the outward trade-off can be a powerful tool, it often introduces expensive special purpose elements into the system. These elements can only be justified if the function that they perform is required frequently enough to make them economical.

REASONS FOR PERFORMING TRADE-OFFS

There are several reasons for performing hardware/software trade-offs:

- to achieve an otherwise unattainable performance goal
- to minimize overall system costs
- to reduce software complexity
- to achieve overall system reliability
- to extend system life
- to improve debugging aids
- to achieve compatibility
- to achieve market position

Achieving otherwise unattainable performance

One of the most common reasons for trading software for hardware is to achieve a performance that could not otherwise be obtained. This process ranges from the inclusion of internal features such as floating point arithmetic and index registers through the addition of specialized processors such as sorters and fast fourier transform processors.[28] These processors may be added to either the I/O system, the memory interface or the CPU. Many of the advanced features of present and proposed computers represent hardware/software or hardware/firmware trades that were made by the manufacturer. Prager[6] gives a good example of a set of trade-offs for improving the performance of the inner loop of scientific computers.

Minimize overall system cost

A frequent goal of designers today is to minimize overall system cost. Thus, it is usually the case that the boundary between software and hardware is drawn in such a way as to minimize hardware costs or even the costs of the entire system, including software. Thus, upward trades are frequently made. They may even be left as an option to the purchaser. Many systems offer optional features such as floating point arithmetic which may be performed either by hardware or by software.[7] Two trends are visible in the marketplace today. One trend is to provide systems in which a large amount of function is being assigned to the firmware in preference to software. The other trend is to develop small fast computers with minimal instruction sets.

Reduce software complexity

A goal which is becoming apparent is to reduce the complexity of both system software and user software by the addition of hardware features which reduce the amount of overall code, provide enhanced run time support, or free the programmer from concerns about limitations of memory space.[8]

To achieve overall system reliability

Software often is subject to failures due to inadvertent over-writing and frequent changes. Thus, there is a tendency of some experimenters to move critical functions to more secure locations. The most secure location is in the firmware or hardware.

Another trend associated with reliability is to move I/O error recovery functions from the software to peripheral controllers or channels.[9]

To extend system life

In the field of computing, the life time of a system is sometimes measured by its ability to change. This adaptability to change is achieved by assigning hardware functions to software or firmware. This phenomenon is particularly observable in communications controllers. However, it is probably an important property of microprogrammed computers with writable control stores.

Improved debugging aids

Monitoring for software errors (such as exceeding the bounds of an array) is very expensive to achieve by means of software alone. However, if the monitoring is built into the hardware it becomes a practical debugging aid.[8] Other hardware aids include firmware monitors, which perform flow tracing, and interrupt schemes, which monitor for violations of system conventions.[10]

The protection hardware, which is a part of many modern computers, is an example of a hardware[11,12] aid to debugging as well as an aid to system reliability.

Compatibility

The design of emulators represents an interesting exercise in hardware, software, firmware trade-offs. An emulator combines hardware, software and firmware for the purpose of executing instructions for a machine other than the machine on which the emulator is run. The selection of the boundary between the three components can significantly affect the performance of the emulator.

Another reason for examining the possibility of hardware/software trade-offs is to achieve intra-line compatibility. When a whole series of computers must be compatible, there are serious constraints that must be placed on the performance of some members of the family. These constraints sometimes limit the performance of downward trades at the large end of the line. The compatibility may be achieved by means of upward trades in the lower performance end of a computer family.

To achieve market position

It is frequently very difficult to demonstrate the cost effectiveness of unique hardware or software features. However, one is led to speculate that a motivation for performing hardware/software trade-offs is to achieve product differentiation and create captive customers who depend on the existence of a unique feature. These customers cannot easily transfer to a different computer.

Marketing considerations have driven many of the minicomputer manufacturers to provide system software that was not required when the first minicomputers were introduced. It is reasonable to speculate that this requirement will stimulate new and creative hardware/software trade-offs for these small machines.

Recurring nature of hardware/software trades

One characteristic of hardware/software trade-offs is that they must be repeated each time a new computer is developed. In fact, hardware/software trade-offs appear at the heart of the design process. They must always be reevaluated in terms of design goals and constraints, as well as within the limits of contemporary technology.

It has been characteristic that hardware/software trades have been performed to achieve high performance in the largest, fastest computers of an epoch or generation. At the same time or slightly later, smaller, more spartan machines without the high performance features are introduced. These machines are optimized for low hardware cost.

During the next epoch the technology evolves so that "advanced" features can be included in new machines at the same price as the smallest machines of the previous epoch. Concurrently, new, even cheaper machines appear without many of the "exotic features." This cycle repeats itself as time goes on.

Inhibiting forces involved with hardware/software trades

While designers would like to make whatever hardware/software trade-offs their imagination and technological constraints allow, they are not always free to do so. Marketing considerations and the cost of developing system software often inhibit this kind of freedom. While there are no formal standards for computer architecture in the United States, manufacturers often impose a standard architecture derived from earlier machines. This permits salvaging of system software and allows users to move from older to newer machines.

The effect of this overriding requirement is that many of the trade-offs that exist in machines today occur as trades between firmware and hardware and not between software and hardware or firmware.

While compatibility with previous systems is an important inhibiting force, it is often relaxed to the extent that the features of an older system form a subset of the features of the newer system. Thus, the older software can usually be used on the new system. However, this implies an inability to use the new features. Thus, even though a machine may include new instructions, there may be considerable expense and delay in making these features available to the user through system software. This expense and delay severely inhibits the ability of designers to freely trade software for hardware and vice versa.

EXAMPLES OF HARDWARE/SOFTWARE TRADES

I/O system

The I/O system in computers has traditionally been an area in which hardware/software trades have been made. Examples of both inward/outward trades and upward/downward trades can easily be found. Some of the reasons for the fertility in this area are:

- A high degree of parallelism is possible.
- The I/O system must deal with a large spectrum of data rates requiring different processing techniques.
- The I/O system is frequently controlled by system software, instead of user software, so that compatibility constraints can be maintained by software rather than hardware interfaces.
- I/O devices seem to undergo a more rapid change than CPU techniques.

The trade-offs that are usually considered lie in the following areas:

- method of transferring data to main memory
- method of monitoring for the completion of an I/O event
- the complexity of an I/O event that can occur between CPU system interventions
- the handling of error conditions

Method of transferring data to main memory

The method of transferring data to main memory depends upon the data rate that must be handled. In the simplest systems, bytes or words of data are deposited in a CPU register. Software is responsible for collecting the data together into main memory size words, transferring the collected words into memory, recognizing the termination of the transmission and analyzing the status of the I/O device. In systems requiring higher data rates, the data is block transferred into main memory by a hardware controller and the software is only responsible for initiating each block transfer and determining the status of the device. The saving in CPU time required can be at least one order of magnitude. In still more complex systems, the software is only responsible for starting a chain of I/O events. These run independently until they are completed.

Method of monitoring for completion of an I/O event

Just as there is a spectrum of techniques for transferring data between the I/O system and memory, there is a spectrum of techniques for monitoring for the completion of an I/O event. At one end of the spectrum the software is required to repeatedly test for completion of an I/O operation. At the other end of the spectrum an interrupt system is used for seizing control of the CPU when an event is complete. The interrupt system may itself offer a range of services which include saving of the machine status, identifying the I/O device which caused the interrupt and providing summary information about the nature of the event that caused the interrupt. All of these interrupt services are subject to hardware/software trades.

Complexity of an I/O event

One of the most important features in determining the amount of software overhead and the amount of software required is the complexity of an I/O event. In the simplest case, the transfer of a single character constitutes an event. In more complex systems, an event may consist of a large chain of block transfers. Devices have been constructed in which extremely complex events can occur as the result of a single command. Examples of such devices are graphics terminals and file processors.[13] Thus, the event may be a lengthy search of a structured file or the sorting of a file. In these cases, the software overhead consists of building an adequately complex command to control the event rather than of monitoring for the completion of the event.

Firmware/hardware trades associated with I/O systems

The trades discussed in the previous paragraphs have all been hardware/software trades. In implementing these trades the designer is also faced with a firmware/hardware trade at all control levels within the I/O hierarchy. In general, the trades are between the same services as discussed above. For example, if an I/O channel is to be implemented using shared CPU facilities,[14] the designer has the choice of requiring the firmware to repetitively check for the completion of an I/O event or to provide for trapping the microcode when an I/O event occurs.

The handling of error conditions

Since errors occur frequently, the handling of I/O errors has usually been the responsibility of software. However, error handling can be the subject of an inward/outward trade-off. Errors are usually detected by some type of a coding scheme which involves examining both the meaningful data and a string of code bits that are transmitted along with the data. This examination can be done by either controller hardware, controller firmware, controller software or CPU software.

The usual strategy in correcting errors that can be detected but not corrected by coding techniques is to repeat the transmission. The initiation and control of this retransmission is also a subject for hardware software trades.[9]

Trades in the CPU

Within the CPU itself there are many design trades that can be made in the hardware/software spectrum. The first group to consider are the downware trades which move function from software to either firmware or hardware. Some examples of these functions are:

- context switching[15,16]
- task dispatching[15]
- register optimization[17]
- memory hierarchy management[18]
- storage protection[12]
- emulation

A second class of trades within the CPU is augmentation of the instruction set for the purpose of simplifying the work of the problem programmer. These trades involve augmenting the architecture to remove constraints or adding of hardware macro functions such as sine or cosine.

The class of trades which remove constraints is frequently associated with address space. These constraints include:

- the size of randomly addressable memory
- the size of the address field in the instruction
- the requirement for instruction and data alignment on word boundaries

Specialized systems

A class of hardware/software trades which is of particular interest is the specialized system. Two types of specialization can be seen in the industry. One class of specialization isolates a function such as sorting, matrix multiplication or fast fourier transform. This may be implemented as a special purpose computer which either operates stand-alone or as a part of a host machine. Some systems have been built or proposed in which a restructurable portion of the system is temporarily configured to obtain high performance for a special function.[19,20] The configuration of these reconfigurable machines is continually changing.

Another class of specialized machine is the machine which is optimized to execute programs written in a higher level language. These machines offer hardware or firmware compilers or translators plus an architecture which is tailored to provide specialized run time support for the functions provided by the language. One characteristic of this architecture is that the command structure is very similar to the constructs of the higher level language for which the machine was designed.

This structure may be markedly different than that of the traditional computer and may be a variant on polish notation or it may be a list or tree structure. Since the internal machine architecture is closely related to the requirements of the source language, the compiler or translator is required to perform much simpler transformations than would be necessary for a more traditional machine architecture. Thus, the compiler or translator is inherently fast. In addition, if the compiler is implemented in firmware, it has the advantage of not requiring main store fetches for instructions and can possibly rely on some parallelism within the CPU.

The execution time support associated with these language specific machines includes specialized instructions, data structures and storage management techniques that are tailored specifically to the language. In the process of providing this support, many functions normally performed by the operating system are moved to the hardware.

Language specific machines have been developed for ALGOL,[3] FORTRAN,[21] EULER,[22] SYMBOL,[23] and APL.[24,25]

Emulators

Emulators are an excellent example of trade-off between hardware, firmware and software. An interesting example illustrating the range of possibilities is the series of 1401 emulators available on several models of System 360 and System 370.

The 1401 emulators on the smaller 360 models are implemented almost entirely by firmware and hardware. Almost all of the 1401 instructions are fetched and executed directly by the emulator microcode. The 1401 emulators on System 370 have a different organization. The emulator firmware implements several instructions which are not 1401 instructions, but which can be used in conjunction with the System 370 instruction set to construct short emulation routines (software) which interpret the 1401 programs. The next 1401 instruction is fetched and decoded by a special emulator instruction at the end of each emulator routine. This instruction forces a branch to the emulator routine that will simulate the next 1401 instruction. Thus, the 1401 emulator on the System 370 has a large software component. The software portion of the emulator also interfaces with OS/360 in such a way that emulator jobs use the normal data management and supervisor services provided by the operating system. Emulator jobs and non-emulator jobs can be mixed indiscriminately.

Figure 1 shows the number of bits of control storage used by 1401 emulators in several System 360 and 370 models.

The System 370 emulators use less control store, but more main storage. The main store is only used, however, when the emulator is in use.

System	Amount of Control Store Used for 1401 Emulation firmware (bits)
360/30	240K
360/40	224K
370/135	109.8K
370/145	38.4K
370/155	38K

Figure 1—Number of bits of control store in 1401 emulators on IBM computers

The 1401 emulator on the 360/40 is illustrative of firmware to software trade. The emulator includes a hardware translator which is used to convert 1401 addresses to physical System 360 addresses. The translation function could have been performed by firmware, but would have required considerably more time.

CONCLUDING REMARKS

This paper has examined some of the reasons for making hardware/software trade-offs and has shown some of the types of trade-offs that have been made in existing machines. Techniques for evaluating trade-offs are discussed in References 2, 26 and 27.

Though hardware/software trade-offs have been carried on throughout the history of computing, the recent introduction of machines that can be microprogrammed by the user should bring about new interest in the topic. Advances in system performance, measurement and modeling are providing better tools for evaluating hardware/software trade-offs and should lead to a more complete understanding of trades.

Language specific machines, intelligent terminals, emulation, machines with firmware operating systems, minicomputers with enhanced capability and implementation of virtual memory will be intensely studied with reference to hardware/software trades during the next few years.

BIBLIOGRAPHY

1 *Processor handbook, PDP11*
Digital Equipment Corporation
Maynard Massachusetts

2 H BARSAMIAN A DECEGAMA
Evaluation of hardware-firmware-software trade-offs with mathematical modeling
Proceedings of the 1971 SJCC pp 151-159

3 *Introduction to the IBM 3705 communications controller*
IBM Corporation White Plains New York Form No GA 272051 1972

4 J F RUGGIERO D A CORYELL
An auxiliary processing system for array calculations
IBM Systems Journal Vol 8 No 2 1969

5 *Control Data 6400/6600 computer systems reference manual*
Control Data Corporation Minneapolis Minnesota

6 D PRAGER
Some notes on speeding up certain loops by software, firmware and hardware means
IEEE Transactions on Computers Jan 1972 pp 97-100

7 *System/360 model 40 functional characteristics*
IBM Corporation White Plains New York Form No 22-6881

8 E A HAVEK D A DENT
Burroughs' B6500/B7500 stack mechanism
AFIPS Conference Proceedings Vol 32 1968 pp 245-251

9 J F KEELEY
System/370-reliability from a system viewpoint

Proceedings of the 1971 IEEE International Computer Society Conference Boston Massachusetts pp 33-34

10 L ROBERTS
Can microcode be used to measure system performance
Proceedings of the 4th Annual Microprogramming Workshop Santa Cruz California September 13-14 1971

11 *IBM system/360 principles of operation*
IBM Corporation White Plains New York Form No GA 22-6821

12 M D SCHROEDER J H SALTZER
A hardware architecture for implementing protection rings
Communications of the ACM March 1972 pp 177-184

13 *2314/2844 multiplex storage control feature-airlines buffer*
IBM Corporation White Plains New York Form No GA 26-5714

14 S S HUSSON
Microprogramming, principles and practice
Chapters 7 and 8 Prentice-Hall Inc Englewood Cliffs N J 1970

15 *MAC computer reference manual*
Lockheed Electronics Los Angeles California
Chapter 4

16 *Sigma 7 reference manual*
Xerox Data Systems El Segundo California

17 R M TOMASULO
Efficient algorithms for expoliting multiple arithmetic units
IBM Journal of Research and Development Jan 1967 pp 25-33

18 *A guide to the IBM/system/370 model 165*
IBM Corporation White Plains New York Form No GA-20-1730 pp 14-24

19 G ESTRIN
Organization of computer systems—The fixed plus variable structure computer
Proc WJCC 1960 pp 33-40

20 W CLARK
Macromodular computer systems
Proc SJCC 1967 pp 335-336

21 T R BASHKOW A SASSON A KRONFIELD
A system design for a FORTRAN machine
IEEE Transactions on Electronic Computers August 1967 pp 485-499

22 H WEBER
Implementation of Euler on the system/360 model 30
Communications of the ACM September 1967 pp 547-558

23 W R SMITH et al
SYMBOL—A large experimental system exploring major hardware replacement of software
Proc of the 1971 SJCC pp 601-617

24 R ZACKS D STEINGART J MOORE
A firmware APL time-sharing system
Proc of the 1971 SJCC pp 179-191

25 A HASSITT J W LAGESCHULTE L E LYON
Implementation of a high level language machine
Proc of the 4th Annual Microprogramming Workshop Santa Cruz California September 1971

26 J D FOLEY
An approach to the optimum design of computer graphics systems
Communications of the ACM June 1971 pp 380-390

27 N R NIELSON
The simulation of time sharing systems
Communications of the ACM July 1967 pp 397-412

28 M J CORINTHIOS
A fast fourier transform for high speed signaal processing
IEEE Transactions August 1971 pp 843-846

A design for an auxiliary associative parallel processor

by M. A. WESLEY, S.-K. CHANG and J. H. MOMMENS

IBM Thomas J. Watson Research Center
Yorktown Heights, New York

INTRODUCTION

The use of highly parallel processing units for computing problems that are highly parallel in structure has been widely studied. The range of systems varies from the duplication of complete processing elements,[1] through the provision of a set of specially tailored small processors attached to a main processor,[2] to the use of cellular arrays;[3] other writers have exploited the inherent parallelism of associative memories as components of parallel processing systems.[4–7]

Associative memories have been proposed either as true content addressable memories,[5] or as processing units.[4–6] In general, for use as a processing unit, each word in the memory, or possibly pairs or groups of words, is regarded as a serial by bit processing unit, all operating in parallel and controlled by a single program. These proposals have included rather complicated control systems to perform bit indexing and other functions necessary to sequence the memory through a program.

An important extension to the concept of associative memories as processing elements was proposed by McKeever,[8] who described the use of three state storage elements with increased logic function at each storage cell; a memory with this feature is referred to here as an associative functional memory. The use of three state cells as a general system technology for conventional sequential processors has been described;[9,10] it is the purpose of this paper to demonstrate that:

1. An associative functional memory with suitable peripheral features could be used to implement many of its own control functions as well as performing processing operations, and could readily be assembled into a complete auxiliary parallel processor,
2. Such a processor would be an attractive means of enhancing the performance of small conventional processors in a wide range of problems.

SYSTEM DESCRIPTION

The associative processor to be described here is intended for use as a programmable auxiliary processor to assist a conventional main processor in special problems. Programs are loaded from the main processor and are used to load data, to process it, and to return results to the main processor. The main processor has at all times the ability to force the auxiliary processor to accept a new program or to branch to a specified location in its program. For applications involving the processing and reduction of very large amounts of raw data, for example, radar signal processing, it would be wasteful to transfer data to the associative processor by way of the memory and channels of the main processor. In these circumstances, the associative processor could be modified to accept data directly from its source, that is, to act as a pre-processor, but would not be expected to exercise control over the data source.

The overall design goals have been simplicity of implementation and generality of application. Simplicity of implementation has been achieved by construction from units which could be standard modules[9] with a minimum of additional special logic, and has led to a potentially fast cycle time. Generality of application has been achieved by implementing many control functions in memory and by the inclusion of some extra associative memory features which are not necessarily required in all applications. The proposed processor consists of two main components (Figure 1): a 1024 word × 64 cell associative functional memory and a 512 word × 50 bit read/write control store. The associative memory is used to store both data being processed and control information. An alternative would have been to have used separate memories; however, the use of a single unit permits the ratio of data to control information to be tailored to any given problem and enables a very simple control system to be used. On the other hand, the single array approach reduces the speed of data processing since many associative

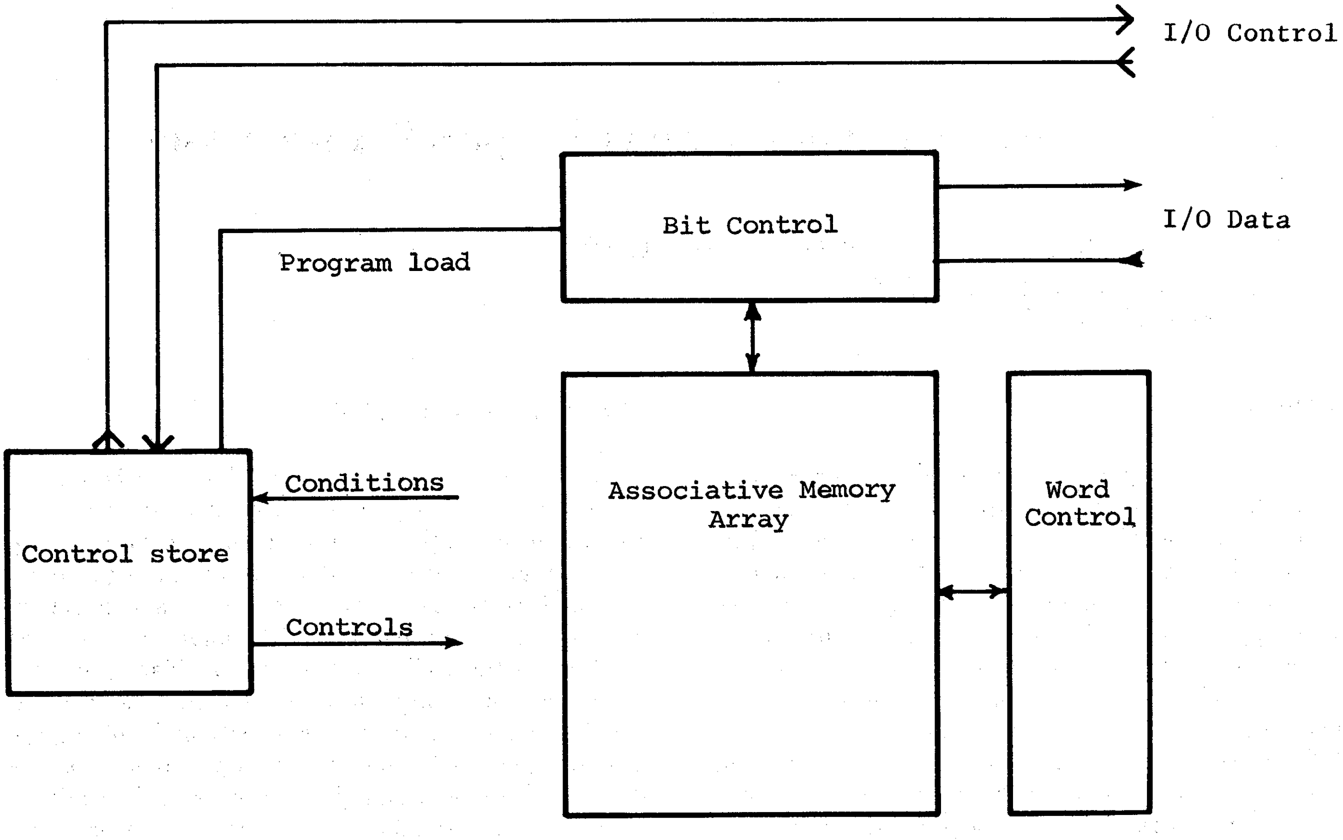


Figure 1—Block diagram of the proposed associative processor

memory cycles have to be used for control operations. It tends to be wasteful in the use of associative cells for control tables, and requires the introduction of extra features to reduce the interference between data and control.

Control sequences for the execution of a program are contained in a read/write control store normally operating in a read-only mode. Conditional branches in the program may be made by testing the condition of various signals in the processor and its I/O interfaces. Program loading, i.e., writing into the control store, is performed under the control of a short, permanent, initial load program.

Input and output data transfers are made by way of the associative array bit control unit. Basic interface control is carried out by the control store which can generate outgoing and test incoming control signals; more complex I/O control, such as an IBM Standard Interface, requires the addition of an interface control unit. Attachment closer to the main processor (e.g., interfacing the main memory) would give higher performance but would imply modifications to the main processor.

Associative processing array

The associative processing array is a two-dimensional array of three state (0, 1, X = "don't care") associative storage cells with arbitrarily chosen dimensions of 1024 words × 64 cells. The array is connected in the word direction to the word control unit and in the bit direction to the bit control unit. In an LSI implementation, the basic module could be a self-contained associative functional memory unit of, say, 128 words, complete with bit and word controls. Modules could readily be extended in the word direction by suitable interconnection of data and control lines; extension in the bit direction may be simulated by software.

Three basic operations may be performed on the array: search, read, and write.

Search

A ternary search argument is generated in the bit control unit between the specified data register (R1, R2) and the specified mask (M, all 1's, all 0's) on a bit by bit basis:

Data

Mask	0	1
0	X	X
1	0	1

X=don't care

Generation of search arguments.

All cells, in parallel, compare their contents with the search argument for that bit column and generate a mis-match signal in accordance with the truth table:

Cell Content

Search Argument	0	1	X
0	0	1	0
1	1	0	0
X	0	0	0

Generation of mismatch signals

Mismatch signals for a cell are ORed to give a mismatch signal for the word; word mismatch signals, in true or complement form, are sent to the word control unit where they may be ANDed or ORed with, or replace the contents of one of two sets of selector latches (P and S).

Read

The contents of a specified set of selector latches (P, S, all 0's) in true or complement form are used to select words to be read. The contents of cells from selected words are ORed in the bit direction onto a read bus (an X state reads as zero) and sent to the bit control unit where they are used to load a specified register (R1, R2, M) based on the value of mask specified (M, all 0, all 1):

Mask

Read Bus	0	1
0	No change	0
1	No change	1

Effect of Read operation on specified register.

Write

Two write commands are provided: Write Normal, and Write Special. In either case a ternary argument is generated in the same manner as a Search argument and acts on the contents of cells in selected words, as defined by the specified selector register in true or complement form (P, S, all 0). The effects on a cell are shown in the two truth tables below:

Write Argument

	0	1	X
Cell Content	0	1	No change

Write Normal

Write Argument

	0	1	X
Cell Content	X	X	No change

Write Special

The word control unit may also perform a one bit shift of a selector register up or down with end around carry, or fill with 0 or 1; a shift takes the same time as an array operation or may be overlapped with an adjacent preceding array operation using the same selector register. This provides the only parallel means of communicating vertically between words. Other writers (e.g., McKeever, Reference 8) have usually specified other operations in the word control unit, such as isolate first match. Although provided by our simulator we have found little use for such operations, which tend to be serial in nature, and for the most part found that they can be economically simulated by software, e.g., by use of a code field. The exception was sorting with an arbitrary number of identical items, when a means of separately identifying multiple matches is necessary.

The bit control unit contains three registers: two data registers (R1, R2) defining a data source or sink for an array operation; and one mask register(M) defining a field for an array operation. Any array operation may use either data register and the mask register, or may replace the mask by a source of all 0's or all 1's. In addition, the control store may specify directly the leftmost four bits each for the mask and data registers. These bits (the immediate field) are ORed into the register outputs without affecting the contents of the register. A non-array operation, a single bit shift operation on any register may be specified; this feature is assumed to take the same time as an array operation unless it is overlapped with an adjacent array operation in which the register being shifted is a data source or sink; again, fill with 0 or 1 may be specified.

Input-output operations

Input-output operations for the associative processor take place through the bit control data register R1. The register is divided into fields each of the same width as the I/O interface data busses. Data may be gated to or from the register under program control and is interlocked with the main processor by interface synchronizing signals. Outgoing interface control signals are generated by the control store and by the run control logic. Incoming interface control signals are either tested as machine conditions by the program, or act directly on the run control logic.

Operation as a pre-processor, taking data from but not controlling another source, would require the ability to transfer into the processor from another interface and generate and test another set of I/O synchronization signals. This modification requires at least two extra bits in the control word and some extra logic, but is not expected to be very difficult to implement.

Control store

The control store (Figure 2) is a conventional (as opposed to associative) read/write store used to hold a program defining the sequence of operations to be performed by the associative memory.[11] During the execution of a program, the control store normally operates in a read-only manner. Each word read out specifies the operations to be performed in the array and also the address of the next program word. The next address may be modified by a condition in the machine, specified by the program word, enabling conditional branches to be made in the program.

The control store contains 512 words of 50 bits, though these numbers may vary, depending on the features included. When formed into groups of mutually exclusive options, the operation options to be specified for the array processing unit fall into rather small groups, so that coding within a group is not very advantageous, and bit significant operation has been chosen. This has other advantages as it increases flexibility and eliminates timing delays through decoders.

It is expected that a semiconductor memory will be necessary to be able to operate at the same speed as the array. Such a memory will have nondestructive read out so that writing into the control store will require special control features. Subroutining capability is provided by a data path to the bit control register R1, enabling subroutine return addresses to be stored in the associative array.

Program loading

Program loading is performed under the control of a small fixed routine held in the first few words of the control store. The program load routine assembles data from the I/O interface into the bit control register R1. This data is interpreted as a control word and the address of the location in the control store into which it is to be stored. The program load routine then gives a special signal "write next cycle" which causes the run control logic to break its normal cycle of read-only operation, and to spend one cycle writing into the control store from R1. Note that the control store data register is not altered and is available for normal operation on the cycle after the write operation is performed. The "write next cycle" control also permits the transfer of programs from the associative array to the control store.

Programming techniques

The guiding principle behind the design of the control system has been to make the hardware simple whilst keeping the system flexible. This principle led to the use of a single associative memory, controlled by a single conventional control store, with both data and control information stored and processed in the associative memory.

Three classes of control information are held in the associative memory:

(1) mask and data register contents for operating on data. In the case of relatively simple operations, such as addition, these register contents are stored in consecutive locations in the sequence in which they will be needed, and are accessed by shifting a selector register reserved for the purpose. In more complicated operations, such as multiplication, where the total number of masks is proportional to p_2 (where p is the field width) and may be large, it may be advantageous to process the masks as data in the manner described in References 9 and 10, and to generate the required sequence of masks; the number of control words now becomes essentially proportional to p.

(2) program flow logic, including counts and logical decisions. These may be programmed directly or, in simple cases, may be implemented by inserting blank words in mask sequences and test-

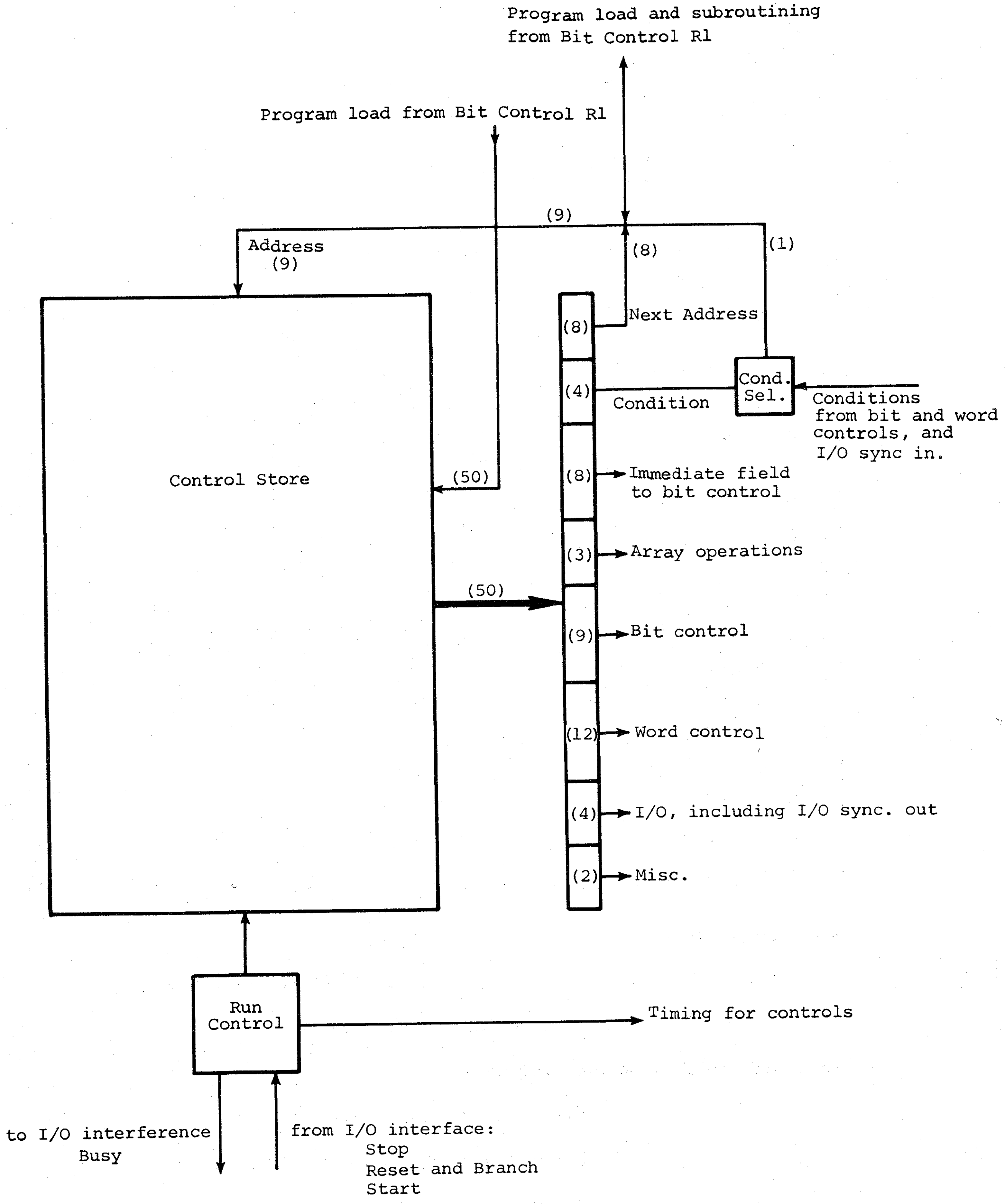


Figure 2—Control store connections

	Immediate Field				Field A						Field B					
R_1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
R_2	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0

0																Data
0																
0																
0																
1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	Control (blank = 0)
1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	
1	0	0	0						0						1	
1	0	0	0						1						0	
1	0	0	0					0						1		
1	0	0	0					1						0		
1	0	0	0				0						1			
1	0	0	0				1						0			
1	0	0	0			0						1				
1	0	0	0			1						0				
1	0	0	0		0						1					
1	0	0	0		1						0					
1	0	0	0	0						1						
1	0	0	0	1						0						
1	0	0	0	0						0						

Immediate field codes: 0 - - - data word

1 0 0 1 start of control sequence

1 0 0 0 control word

Immediate field allocation for data words:

bit 1 0 : data word

bit 2 0/1 : not active/active marker

bit 3 0/1 : carry 0/I

Figure 3a—Memory organization of addition: $A' = A + B$

Location	Operation	Immediate Field	Selector	Data Source or Sink	Mask	Next Address	Comments
14	Search	1001	P	-	0	15	load R1, R2, M. keep mask table pointer in P.
15	Read	0000	P, shift down	R2	1	16	
16	Read	0000	P, shift down	R1	1	17	
17	Read	0000	P, shift down	M	1	18	
18	Search	0---	S	-	0	19	reset carry and active markers in data words
19	Write	-10-	S	-	0	20	
20	Search	010-	S	R2	M	22	identify no change combinations and mark as inactive
22	Search	011-	OR into S	R1	M	23	
23	Write	00--	S	-	0	24	
24	Read	0000	P, shift down	M	1	25	read new mask
25	Search	01--	S	R2	M	26	identify a field bit changing to 0; update
26	Write	001-	S	R1	M	27	
27	Search	01--	S	R1	M	28	identify a field bit changing to 1, update
28	Write	000-	S	R2	M	29	
29	Search	0---	S	-	0	30	set active markers
30	Read	0000	P, shift down	M	1	31	read new mask
31	Write	-1--	S	-	0	20/21 (M=0)	test for mask = 0
21							

Figure 3b—Program for addition: $A' = A + B$

ing for an all zero read out. Note that the only internal condition tests available to the programmer are zero tests on the bit and word registers; an alternative would have been a test on a single bit.

(3) partitioning. The immediate field provides a fast software technique for partitioning the single array into groups of words. The four bits of the field permit 16 interleaved partitions of arbitrary size. This feature is particularly valuable for distinguishing and separating data and control information; for example, a 0 in the leftmost bit position may signify data while a 1 signifies control.

A further consequence of the use of a single array is the need to load and store the mask register from and to the array. The three array operations have been generalized for this purpose.

Programming example: Serial-by-bit addition

This example is given to show:

(1) the use of the immediate field
(2) the use of the associative array for both data and control information
(3) the ability to define fields independently of the program by means of control tables.

Suppose we wish to perform the addition of two fields, A and B, the result to overwrite field A, i.e., $A'=A+B$. The minimum possible number of array operations per bit is 6 (4 Search and 2 Write); however, this assumes no performance loss handling control operations. The addition algorithm given below takes 11 operations (9 if the inner loop is expanded to handle two bits consecutively). The algorithm uses $2p+3$ memory words to store masks and data register contents (p is the field width); we have found that, in general, it is possible to trade less speed for less control storage.

The algorithm is illustrated in Figure 3. The first six instructions locate the start of an addition control table, load the two data registers R1 and R2 with constants which remain unchanged throughout the algorithm, load an initial pattern into the mask register, and initialize the immediate field. Three bits of the immediate field are used: bit 1 indicates data or control words, bit 2 is an activity marker used to indicate whether a word has been completely processed in the current bit position, and bit 3 is a carry and is initially zero.

Instructions 20-31 make up the main loop of the algorithm which proceeds in a serial by bit manner starting with the least significant bit. At each bit position the no change condition in the A and carry bits is detected, and these words are marked as inactive. The remaining words are tested for changes in the A field and are updated. Indexing across the fields is achieved by the mask register contents, which are read sequentially from the control table. Execution of the loop ceases when an all zero mask is read out.

APPLICATIONS

The principal mode of parallel processing employed in this associative processor is serial by bit, parallel by word, over some selected subset of words in the memory. Thus a memory of 1024 words has a potential processing parallelism of up to 1024. Operating in a serial-by-bit manner across fields inherently requires more cycles than a conventional machine with bit parallel processing. This is particularly significant in arithmetic operations; for example, 16 bit addition requires about four times as many control cycles as a System/360 Model 30, 32 bit addition requires about eight times as many, and this must be more than cancelled by the parallelism used. At present we are limited to fixed or block floating point operation; normalization in general floating point is prohibitively time consuming. In bit manipulation operations, the programmable field feature (i.e., the ability to define fields by mask control tables stored in the associative memory) may enable the associative processor to take fewer operations than a sequential machine.

The overall performance of the associative processor is affected by a number of overheads. It is assumed that the processor would be used for repetitive execution of a program, so that program and control table loading times need not be included in the problem-solving time. Input and output of data is sequential by word and can be very significant. In general, the processor as described with a single I/O data path is only suited to problems with a high processing to I/O ratio; however, multiple I/O data paths could be provided to each of a number of partitions. After each stage of parallel computation (e.g., after a vector addition) it is generally necessary to reorganize the data for a subsequent stage of processing; this too can use significant amounts of time and must be minimized by careful algorithm selection and memory organization.

The performance of the processor has been studied with the assistance of a very flexible simulator program which allowed function truth tables to be defined at object time. Execution times, including processing, input/output, and data reorganization, have been computed assuming a cycle time of 100 nsec, which is believed to be within the capability of an LSI technology.

A wide range of examples have been studied for the associative processor and are discussed here without details of programming techniques. The aim in choosing examples has been to investigate the versatility of the associative processor and to demonstrate its performance on problems for which special purpose processors are being built. The examples are summarized in Table I; performance figures for the associative processor are based on a cycle time of 100 nsecs and an I/O data rate of 1.5 μ secs per byte.

Picture processing

The functional memory may be regarded as a two-dimensional array of storage cells. Given a memory with suitable dimensions, two-dimensional pictures may be stored in two-dimensional form and, since neighboring point relationships are preserved, local processing operations may be performed directly and with a high degree of parallelism. Analog picture element values may be coded into a number of adjacent bits in either the bit or word direction; pictures too large for the memory may be partitioned and processed in separate pages, but this requires care in piecing the edge results together.

As an example, consider the application of a two-dimensional binary mask operator ($n_x \times n_y$) to a binary picture ($N_x \times N_y$) stored in the functional memory. The algorithm proceeds by searching sequentially for each line of the operator, centered on one column of the picture. The result of the first search operation is loaded into a selector register and shifted one position; the results of subsequent searches are ANDed into the previous selector register contents before shifting. After n_y search operations, the selector register contains the full result of applying the operator to the column and may be either stored back into the memory or output; further columns may be processed sequentially. With $N_x = N_y = 144$, application of 25 operators with $n_x = n_y =$

7 takes 120 milliseconds and is estimated to be 610 times faster than a 360/30. Note that this problem gains performance through both the parallelism of the associative processor and its ability to tailor data fields to the needs of the current algorithm.

An alternative approach, suitable for on-line character recognition, would be to exploit the symmetry of the picture-operator system and hold the mask operators in the memory and search them with the picture as received from a scanner. This operation is the "feature extraction" process of character recognition; the resultant feature vector may subsequently be matched against a stored library of standard reference feature vectors; in both operations the three call states may be used to represent ternary data. Distance measures between the feature vector and all the reference vectors may then be computed in a serial-by-bit manner; the recognition process may be completed by testing for the minimum distance using parallel search techniques.

Lewin sorting algorithm

The Lewin sorting algorithm[12] was originally proposed for an associative memory with a special hardware feature to indicate whether a column contained all 0's or all 1's. This feature may readily be simulated by software on this processor; for example, searching for 1 on a data column and a subsequent read of a marker column containing all 1's will indicate whether or not the data column contains all 0's. An all 1's condition may be similarly detected.

The algorithm finds, for example, the largest of a set of numbers by searching for columns containing a mixture of 0's and 1's. If no such columns exist, all the numbers are identical and are equal to the largest one. Otherwise, the leftmost mixed column is searched for numbers with 1 in this position, and the operation is repeated on this new subset.

TABLE I—Summary of the Performance of the Associative Processor

	Distribution of total processing time				
	Processing	Data Reorganization	I/O	Total Processing Time	Speed Up Over 360/30
Picture Processing	97%	—	3%	122 millisec.	610×
Sorting	70%	—	30%	20 millisec.	110×
Matrix Mult.	31%	7%	62%	1 millisec.	78×
Fourier Transform	17%	44%	39%	31 millisec.	75×
Hadamard Transform	4%	46%	50%	12 millisec.	79×
1-D Filter	40%	—	60%	10 millisec.	280×
2-D Filter	50%	—	50%	20 sec.	510×

The number of operations taken by the associative processor to execute the algorithm is very data dependent; worst-case figures are given in Table I for an internal sort of 1000 items using 16 bit keys and show a speed up of a factor of 110 over a 360/30.

As mentioned previously, a sort of identical items requires a means of isolating the components of multiple matches; in this example, where an arbitrary number of identical items may be present software techniques require a wide code field and are therefore expensive. We have assumed the existence of an isolate first hit feature.

Tree searching

One of the major problems in artificial intelligence is to perform efficient tree searching. Since the number of nodes of a tree grows exponentially with respect to the depth of the tree, the tree searching time also increases exponentially, rendering deeper search impractical. It is clear that in tree searching the same sequence of computation and condition testing is performed on every node. Thus the basic requirement of "Single Program Multiple Data" processing is satisfied and we can perform computations upon all nodes in parallel. The tree may still have to be grown step by step, but this is probably unavoidable.

It is difficult to define a typical tree searching problem and, since performance of both the associative processor and a conventional processor are highly problem dependent, no performance comparisons are given. However, we note that the performance improvements in the region of 2–3 orders of magnitude have been found in simple game-playing problems.

Matrix operations

Many matrix operations are inherently parallel in nature and may readily be programmed for the associative processor. Vector addition, subtraction, and multiplication operations, and summation of elements of a vector, may be executed very efficiently; division may be performed only with difficulty. Thus, matrix multiplication is very attractive, but operations involving a high proportion of divisions is not likely to show any great advantage on the associative processor. When only a small number of divisions are required, they may be performed by the main processor (e.g., pivotal element normalization in matrix inversion[2]).

Fixed point multiplication of 10×10 matrices at 16 bit precision gives a performance improvement of 78 times. Larger matrix sizes may be partitioned to fit the processor and show approximately the same processing performance improvement because I/O time dominates.

Fast Fourier and Hadamard transforms

The fast Fourier[13] and Hadamard[14] transforms are closely related operations used particularly in signal and image processing. The radix-2 fast Fourier transform computes the Fourier transform of a set of points $A_1^0 \cdots A_n^0$ by means of a sequence of transformations $A^0 \rightarrow A^1 \rightarrow \cdots A^{m-1}$, where $n=2^m$. Each of these transformations is made up on $n/2$ pairs of elementary operations of the form

$$A_p^{i+1}=A_p^i+W_p^iA_q^i$$
$$A_q^{i+1}=A_p^i-W_p^iA_q^i$$

where W_p^i is a complex 2^{i+1}th root of unity in the fast Fourier transform, and 1 in the Hadamard transform. Each of the pairs of elementary operations in a transform may be performed in parallel and consists of a complex multiplication followed by a complex addition and subtraction; the result of a transformation may overwrite the input to the transformation.

Many algorithms have been proposed for the selection of the pairs of indices p and q. The procedure chosen for use here selects the indices in a regular manner and allows efficient use to be made of the select latches as a means of parallel communication between words. For the first transformation $A^0 \rightarrow A^1$, A_p and A_q are $n/2$ words apart; for $A^1 \rightarrow A^2$, $n/4$ words apart, etc. However, this procedure has the disadvantage that if the input data are in order, the results will be permuted with their addresses in bit reversed form, though this may be corrected when the results are transferred back to the main processor. The basic steps of the algorithm have already been described[15] for an associative processor with external storage and separated data and control functions.

The implementation of a useful size of Fourier transform within this associative processor requires the use of a larger memory array. The principal reasons are the need to store the complex roots of unity and the inclusion of an address field to enable blocks of operands to be identified rapidly. A 1024 point complex transform with 14 bit precision may be fitted into an associative memory of 1273 words of 89 bits with a performance approximately 75 times faster than a 360/30.

The Hadamard transform may be regarded as a square wave analog of the sine and cosine wave Fourier transform and has many advantages from a computational point of view. In particular, the use of square waves of amplitude±1 makes multiplication unnecessary, and an ability to generate square wave transition lengths for a transform of length 2N from a transform of length N removes the need for a stored table of coefficients. The Hadamard transform also has a fast Hadamard transform algorithm. Performance on the associative processor for a 1024 point real transform is shown in Table I. Note that in both these transforms data reorganization becomes very significant.

Digital filtering

Digital convolutional filters of the form:

$$y(t)=\sum_{\tau=1}^{n} x(t-\tau)g(t)$$

where $g(t)$ is a filter of length n

$x(t)$ is the filter input

$y(t)$ is the filter output

may be implemented on the associative processor in a number of ways, the choice depending principally on the dimensions of the problems, e.g., filter length, data record length, and number of filters. The most efficient method, in the sense that I/O operations are minimized, is to store the filter vector g permanently in the memory and to regard the data points as scalar inputs operating on all elements of the filter vector. This method is applicable when $N \approx$ number of words in the memory, where N represents either a single long filter or a number of shorter filters of equal length. Note that, in the Single Program Multiple Data form of parallel processing, a scalar operation on a vector differs from element operations between a pair of vectors in that it is now possible to perform look-ahead operations when processing the scalar quantity, thereby approximately halving the execution time.

The algorithm assumes that the memory is partitioned into two fields of equal size, one for the filter vector g and the other for partial results. Processing proceeds in a pipeline manner—a new data point is received and used as a scalar multiplier on the filter vector, the products being added into the adjacent partial result field. The partial results are shifted one word position and the process repeated with the next data point. After the first n data points have been processed, one output result will be available for each filter held in the memory; thereafter, output results are available after each new data point has been processed.

An alternative method to be used when the filter is short is to load the memory to capacity with data points and to apply the filter coefficients as external scalars. When the whole filter has been applied, all the results may be read out. The processing time for this method is the same as that for the stored filter, but the I/O time is significantly greater.

Two examples have been considered, both using the stored data method. The first is a typical seismic signal processing problem and has a 1000-point data record, a 25-point filter, and operates at 16 bit precision. The second is a picture processing problem similar to that posed by Mariner pictures with a picture of 600×684 elements, a two-dimensional filter of 15×15 points, and operates at 8 bit precision.

The results are shown in Table I. In spite of the I/O overheads, the performance improvements are large; in particular, the space picture processing performance reflects the ability of the associative processor to tailor its field lengths to the problem.

Convolutional decoding

In this example, the associative processor is used to perform error-correcting decoding operations. The Viterbi decoding algorithm[16] is given as an example; however, in order to understand the decoding algorithm it is necessary to first describe the coding process.

The encoder has the canonical form of a shift register of length S. Each time an information digit is encoded, the contents of the shift register are shifted right, the rightmost bit being discarded, and the information digit is stored in the leftmost bit of the register. The encoded message bits are the modulo 2 sums of some bits in the shift register; the ratio of information bits to encoded message bits is known as the code rate.

The present contents of the shift register may be regarded as the state of the encoder, and a state transition diagram may be constructed for every input of an information bit. The Viterbi decoding algorithm is based on storing, for each possible state of the encoder, a history of the most probable, in some sense, sequence of information digits to reach that state. Each state and its history has a distance measure associated with it. When a new set of encoded message bits are received, the histories are updated by computing the error distance between the received bits and the true bits corresponding to each state transition, and adding this to the distance measure for the corresponding history. The histories are arbitrarily restricted to a length $3(S-1)$ and, after each updating, the bit $3(S-1)$ bits away in the history with the lowest distance measure is output as a decoded bit.

A number of examples have been studied with various values of S and code rate. Comparisons have not been made with a conventional machine because special purpose processors are being built for these decoding problems. For $S=6$, rate$=\frac{1}{2}$, the associative processor takes 100 μseconds per bit, i.e., 10K bits/sec. For $S=9$, rate$=\frac{1}{3}$, the processor takes 370 μsecs. per bit, i.e., 2.7K bits/sec. This variation in performance with shift register length S is almost entirely caused by an increase in data reorganization overheads caused by a larger number of encoder states.

CONCLUSIONS

The auxiliary associative processor described in this paper has been shown to have a high performance on a wide range of problems which are inherently parallel in structure. The major drawbacks have been found to be in the processor's ability to handle only fixed point or block floating point arithmetic, and the difficulty of performing division. The principal system problems have been in the operating overheads of I/O and data reorganization. The I/O overhead could be reduced by integrating the associative processor into the main processor, which would also permit more complex interaction between the processors or by providing multiple I/O paths. The data reorganization overhead is caused mainly by long shift operations in the selector latches; these could be reduced by hardware and/or software partitioning of the memory, enabling inactive blocks of words to be by-passed. In Reference 17 the data reorganization problem is studied in detail.

The consequences of using a single array to hold both data and control information are hard to isolate. In operation time the overhead for control operations is always less than 50 percent (arithmetic operations) and is generally much less. In memory space, the price is at least one bit of each word (the immediate field) and up to 25 percent increase in size (Fourier transform). In contrast, the flexible partition between data and control, and the ability to tailor fields to the problem have proved very powerful.

The decision to use the three-state cells of McKeever[8] was based on the aim for generality of application. In parallel binary arithmetic operations only two of the three states have been used; however, the third state has been used for data representation in picture processing and tree searching, and for implementation of control functions in all examples. In practice the three-state cell may be implemented by 2 two-state cells; the possibility then exists of having two-state cells individually for 2 state operations, and in conjunction for larger numbers of states. In this paper we have not pursued such an approach.

Economic realization of the processor requires the availability of high performance, low-cost integrated circuit technologies. However, the system design has aimed at the use of only a small number of different components, most of which are memory rather than random logic. The components used could be a standard technology suitable for both conventional and parallel systems.

The performance figures quoted in this paper have been obtained with the aid of a software simulator at the microprogram level. Little work has been done on the development of a higher level language or assembler for the processor.

ACKNOWLEDGMENT

We are indebted to Dr. R. Lyons for drawing the Lewin sorting algorithm to our attention and pointing out its suitability for execution on an associative functional processor.

REFERENCES

1 D L SLOTNIK W C BORCK
R C MCREYNOLDS
The Solomon Computer
Proc FJCC pp 97-107 1962

2 B A CRANE J A GITHENS
Bulk processing in a distributed logic memory
IEEE Trans on Elect Computers Vol EC 14 pp 186-196 April 1965

3 J H HOLLAND
A universal computer capable of executing an arbitrary number of sub-programs simultaneously
Proc FJCC pp 108-113 1959

4 G ESTRIN R FULLER
Algorithms for content-addressable memories
Proc IEEE pp 118-130 Pacific Computer Conf 1963

5 R G EWING P M DAVIES
An associative processor
Proc FJCC pp 147-158 1964

6 R H FULLER R M BIRD
An associative parallel processor with applications to picture processing
Proc FJCC pp 105-115 1965

7 J A GITHENS
An associative, highly parallel computer for radar data processing
Parallel Processor Systems Technologies and Applications editor L C Hobbs pp 71-86 Spartan Books 1970

8 B T MCKEEVER
The associative memory structure
Proc FJCC pp 371-388 1965

9 M FLINDERS P L GARDNER J G MINSHULL
R J LLEWELYN
Functional memory as a general purpose systems technology
1970 IEEE Computer Group Conference June 1970

10 P L GARDNER
Functional memory and its microprogramming implications
IEEE Trans on Computers Vol C20 No 7 pp 764-755 July 1971

11 D A SAVITT H H LOVE
Association storing processor study
Hughes Aircraft Technical Report No TR-66-174
(AD 488538) June 1966

12 M H LEWIN
Retrieval of ordered lists from a content addressed memory
RCA Review June 1962 pp 215-229

13 G-AE SUBCOMMITTEE ON MEASUREMENT CONCEPTS
What is the fast Fourier transform
IEEE Trans Audio and Electroacoustics Vol AV-15 pp 44-55 June 1967

14 W K PRATT J KANE H C ANDREWS
Hadamard transform image coding
Proc IEEE Vol 57 No 1 Jan 1969 pp 58-68

15 M A WESLEY
Associative parallel processing for the fast Fourier transform
IEEE Trans on Audio and Electroacoustics Vol Au-17 No 2 pp 162-165 June 1969

16 A J VITERBI
Error bounds for convolutional codes and an asymptotically optimum decoding algorithm
IEEE Trans on Inf Theory April 1967 Vol IT-13 No 2 pp 260-269

17 S K CHANG
Parallel computation of local operations
Proc Third ACM Symposium on Theory of Computing May 1971 pp 101-115

An eclectic information processing system*

by R. CUTTS, J. HAYNES, H. HUSKEY, J. KAUBISCH, L. LAITINEN, G. TOLLKUHN, and E. YARWOOD

University of California
Santa Cruz, California

INTRODUCTION

'This paper is a progress report on a computer system which is now being designed and constructed. As the title indicates, ideas that seem good have been taken from many different sources. Many features of contemporary large systems that were earlier incorporated into a plan for a large machine[1] are now being applied to this smaller system.

The new design is grounded in hardware string processing, affording a greater generality of application than is typical in existing small systems. The structure employs multiple, specialized sub-processors operating concurrently. Interrupts are not needed; rather the natural breakpoints in evaluating expressions cause processors to move from one task to another under the control of hardware queues. Main storage is allocated in variable-length segments. Parsing hardware facilitates the use of input languages of familiar style. Disk storage is provided only for currently-active users; long-term storage of user data is on personal magnetic tape cassettes.

The system structure readily admits the addition of evaluating processors designed to improve performance in specific areas. Vector arithmetic and graphical display of functions are two examples in the prototype system. Aside from its computational capabilities, the system can serve as a communications "front end" for a large computer system.

The goal is a relatively inexpensive system that will serve the needs of a diverse community of users, such as the faculty and students of a small college. These potential users cannot foresee their information processing needs any better than they can foresee the results of their yet-undone research. This means that the only completely satisfactory system is one that is universal, i.e., one that can compute anything that is computable.

A Turing machine, although universal in the required sense, is far too cumbersome. Thus, it becomes necessary to choose for implementation a set of primitive processes that is a compromise among generality, complexity, and speed. Any particular operation might be built-in or composed from more primitive operations. Except for speed the user does not know the difference; and since the primitives are universal he can always use them to build other high-level operations to suit his needs. In other words, an extensible system is wanted. Of course, there are other desiderata for a computer system besides extensibility and universality, such as conciseness, naturalness, ease of learning, self-documentation of programs, ease of generating efficient runtime representations, etc. Instead of concentrating on just the programming language, this project aims to build an integrated language and logical structure for computing.

Recursive string processing is the basis of this system. This is a general technique; most computer applications can be characterized as operating upon an input string to produce an output string as specified in a procedure string. Properly outside this scope are computer graphics applications, since these apply to two-dimensional objects rather than to strings. However, as usually instrumented, even graphical processing begins by transforming a picture into a string of point values, and ends by stringing point values together for the output device.

It is fair to question the wisdom of designing a timesharing system in the light of the one-user free-standing computer-on-a-chip promised by MOS technology. There are really two questions: (1) Are there resources which can usefully be shared among users? (2) Can resource-sharing be implemented economically? The authors offer an affirmative answer to both of these questions. The first resource worth sharing is high-speed storage. Users have varying storage requirements, so that with individual machines of any storage size there

* This work was supported by the National Science Foundation, Grant GJ-30436X.

would always be storage going to waste in some machines while other users could not run for lack of storage. The economy-of-scale argument to justify resource-sharing, while no longer convincing as far as hardware cost is concerned, remains valid in considering the incremental cost of adding users to the system. For individual machines the cost of serving N users is N times the cost of one machine. For N users of a shared system, the cost per user may exceed the cost of individual machines for small N, but for large N the cost per user tends to decrease. This comes about for two reasons: (1) Adding one user without increasing system resources does not seriously degrade service to the other users. (2) The peaks in moment-by-moment demand for resources tend to average out, so that only the total average demand must be satisfied rather than the sum of the peak demands. String processing is characterized by an unusually dynamic pattern of storage requirements, since the objects of computation frequently vary in length.

It is less clear that processor time is worth sharing since the price of processor logic has dropped by several orders of magnitude; while the cost of software to keep the processor busy has increased considerably. In this system the problem is largely avoided by employing a number of dissimilar, cheap sub-processors operating simultaneously; and by taking advantage of natural breakpoints in execution to switch a processor from one user to the next. The several processors are driven by hardware queues. Interrupt control of them is not needed, because any process requiring service simply makes an entry in the appropriate queue.

The economics of hardware still make it worthwhile to employ more than one level of storage in a system. Currently, rotating memory remains the choice for secondary storage. The difference between the speed of fast storage and the average access time of rotating storage is a severe problem. If requests for disk transfers are handled on a first-come, first-served basis, one average access time is required for each transfer. Aside from slowing down user processing this is a gross waste of memory residence time. A block of data in memory sits around for several milliseconds awaiting disk access and is then read or written very quickly. Utilization of fast storage can be improved considerably by employing a "smarter" disk controller. For a disk-to-main transfer the controller can delay seizing memory until just before transfer starts; and for a main-to-disk transfer the controller might write the data into the first available location rather than waiting for some location specified by the processor. It is now well-known that a disk controller should schedule its own tasks according to a shortest-seek-time-first policy.

USER LANGUAGE

To construct a computer system like the one contemplated, one typically selects an existing general purpose computer, then selects, or designs, a user language, providing appropriate extensions to the language to give the user access to all functions of the system. Next an interpreter is implemented for the user language in the machine language of the selected computer. In this project, however, a general and extensible user language is being designed first, after which the set of small processors to interpret the language will be designed and constructed.

In recursive string processing the primitive element dealt with is the variable-length character string. Numbers, identifiers and subroutines, as well as arbitrary strings of text, are all examples of this type. The result of evaluating a function may immediately be used as input for further evaluation. Examples of earlier recursive string processing systems are Calvin Mooers' TRAC®[2] and Christopher Strachey's General Purpose Macrogenerator.[3]

The simplicity of the scanning algorithm for the TRAC language follows from the fact that the source code is already in a form of prefix-Polish notation. The Polish string is the proper starting point for the design of a machine language (see Barton[4]). This notation makes TRAC somewhat awkward as a user language. The system described here provides a more conventional user language called ZIP, using infix operators with operator precedence, function notation, and other such attributes, but giving the user access to a set of string manipulation primitives like those of TRAC. Such a notation is clearly desirable, especially for arithmetic computation, allowing one to write

```
A=B+C*2-(5+C);
```

or S=CONCAT(S, [THE HOUSE.]);

instead of (the equivalent TRAC expressions):

```
#(DS, A, #(SUB, #(ADD, #(CL, B), #(MUL,
   #(CL, C), 2)), #(ADD, 5, #(CL, C))))
```

or #(DS, S, ##(CL, S) THE HOUSE.)

Note that in ZIP one may refer to the value of an identifier without marking that identifier with the CL (call) function, as is necessary in TRAC. On the other hand, quoted strings must be marked by square brackets in ZIP, where no such quoting is necessary in TRAC.

To implement such a language, there is a hardware processor, called the parser, to take text from the source string and place operands on an evaluation stack in the correct sequence for evaluation by the evaluator-processor. The interpretation of source text is accomplished by alternating actions of the parser and evaluator. The parser places operands on the stack until an operation is called for, then signals the evaluator to perform that operation, after which control is returned to the parser. In addition to the source string and the evaluation stack, each user will have a parse stack, to be used exclusively by the parser in rearranging operators from the source string. The basic algorithm for infix-to-postfix-Polish translation using a stack is discussed thoroughly in References 6, 7, and elsewhere. In our interpretive scheme, operations are performed at appropriate steps in the parsing of source code, rather than operators being placed in a postfix-Polish code string for later execution. In either case the algorithm for translation from infix notation is the same. By keeping these two parts of the system, the parser and the evaluator, conceptually and physically separate, either the language or the evaluator primitives may be redesigned without extensive global design changes.

Implementation of this parser in hardware may seem a formidable task, but the translation process will be simpler than in conventional compilers because of the direct correspondence between the source-language operators and the primitive operations of the evaluation processor. An example of this is the if-then-else construct, which could be of the form:

IF ⟨logical expression⟩ THEN [⟨statements⟩]
 ELSE [⟨statements⟩];.

Instead of emitting test and branch instructions into object code, as would a typical compiler, our interpretation scheme need only present the evaluator with the following top-of-stack configuration:

```
| logical value | string of text | string of text |
                                                  ↑
                        top of evaluation stack───┘
```

and call for the if-then-else operation. This causes the evaluator to select the first or second string of text, depending on the value of the logical expression. The selected string is then copied into the source string and interpreted.

THE PROCESSOR SYSTEM

In order to simplify this presentation, a five-processor system will be described. Whereas economics forces one to use several levels of memory (e.g., high speed, disk, and magnetic tape), this presentation will be in terms of a single level.

The processors are independent hardware devices all referring to the same memory. They are:

1. Input processor
2. Output processor
3. Parser
4. Evaluator
5. Allocator

Each processor has a request table with an entry corresponding to each user. Each processor scans its table in round-robin fashion and performs tasks for any user for which a request exists.

The input processor

The processor scans for input characters from each user (actually from each input terminal on the system). It has two tables; one indicates for each user whether input is expected, and the second specifies whether input is to be echoed on the user's output device. If an arriving character is expected then it is placed on the user's evaluation stack. If it is a terminator symbol (found by checking the character against the list of terminator symbols for that user), then input expected is set to zero for that user, and his entry in the parser request table is set. If echo is "on" the character is displayed.

If input is not expected, the character is compared with a list of special symbols such as "log-on", "suspend" (pause), "continue" (start up after pause), "kill" (stop doing everything), etc. If it is none of these it will be ignored.

Log-on causes initialization of pointers and the assignment of initial segments for stacks. Suspend disables a user and no processor will take any action for him. Continue re-enables the user and processing continues. Kill clears requests in all tables and initializes that user.

The parser

The parser scans its table for requests. If a request exists for a particular user the parser starts obtaining characters from that user's source string. An item from the source string is compared with the top of the parse stack, and depending upon a precedence table, the item may be (1) discarded, (2) placed on the parse stack, or (3) placed on the evaluation stack. Marks are automatically placed in the evaluation stack to delimit multicharacter items. If an operation is to be performed then the evaluator request table is marked and the parser request table entry for that user is cleared.

The evaluator

The evaluator scans its table and for a given user's request performs the operation specified on the top of the evaluation stack. When the operation is complete the parser request table is again marked for that user. Thus, for each user the parser and the evaluator are alternately processing text and performing operations.

The evaluator does the conventional arithmetic operations, string operations, etc. Named operands are kept in a linked list of segments called the form store. A form segment contains both the name of an operand and its value. To fetch an operand the evaluator searches the user's form store, and upon finding the form name copies the corresponding value.

To store an operand, form store is searched to find any previous instance of the name. The allocator is called to delete this segment (the memory space is returned to the free list). Then the allocator is requested to provide space for the new form. The name and value are copied from the evaluator stack into the new space.

The allocator

The allocator answers requests to release memory and to obtain memory. The release process involves connecting the released segment into a list of free segments. If either of its neighbors is free merging takes place to produce the largest possible contiguous free segment. Obtaining memory for a user involves (1) scanning the free list to find a segment long enough, (2) disconnecting it from the free list, and (3) returning the address of the segment to the user.

STORAGE ALLOCATION

Main storage is addressable by byte, and is allocated to processors in variable-length segments. While segmentation is more difficult to implement than some other storage allocation schemes, and while it tends to tie up some storage for bookkeeping, it is desirable in a system in which all data are variable-length strings. The smallest possible segment is nine bytes, as a not-in-use segment contains this much linkage information. An in-use segment typically has four or five bytes devoted to linkage. The largest segment can, in principle, be as large as all of storage; but, in practice, the segments that can be assigned to processors are restricted in size.

When the system is initialized, storage is partitioned into two segments. The segment beginning at address zero is called the base segment, and contains system global information. The remainder of storage is formed into a single free segment. As the system runs, processors request and release space, causing the free storage to become fragmented. The free segments are doubly-linked together into a chain. In-use segments are chained to various lists belonging to individual processors. These lists include the stacks and form store. The formats of segments vary, but all contain the extent of the segment in the first two bytes.

The allocator is a processor which has the task of managing the free storage chain while servicing processor's requests to obtain or release storage. All such requests are made to the allocator. A reserved location in the base segment contains a pointer to the beginning of the free chain. Another reserved location contains the total amount of free space currently available, ignoring fragmentation. When a processor requests additional space the allocator checks whether this much space is available at all. If so, it begins a search of the free chain for a free segment at least as large as requested.

The free list is not ordered in any particular way. In searching for space the allocator chooses the first segment that is big enough rather than looking for a particularly close fit. This policy is one of Knuth's[5] recommendations; a best-fit takes more time than a first-fit policy and tends to proliferate small free segments that are rarely useful. When the allocator finds a sufficiently large segment there are two possibilities: the segment may be large enough to satisfy the request with a usable amount of space left over, or the segment may be an exact or close fit. In the former case, space is excised from the tail of the free segment and formed into a new segment. In case of a close fit there is not enough space left over to form a free segment, so the entire segment is taken from the free list; the extra bytes, if any, are marked null. If the request can be satisfied the address of the segment is returned to the calling processor; otherwise it moves on to another task with the request unsatisfied. When a processor is ready to release a segment that has been in use it calls the allocator with a pointer to the segment. The allocator adds the extent of the segment to the count of free storage and proceeds to connect it with the free chain. First, its neighbors are checked; they might also be free. If so, the segment being released can be merged into its free neighbor(s). Otherwise, the segment is simply added to the end of the free chain. Merging whenever possible reduces fragmentation. Knuth's experiments suggest that fragmentation will not become a serious problem so long as segments are restricted in size to less than perhaps 1/10 of total storage capacity.

The allocator may satisfy requests for storage only from the free chain. The system will keep a tally of the total storage assigned to each user so that the performance can be monitored. The amount of storage allocated to an inactive user is only a couple of bytes in the base segment. When a user becomes active he is assigned segments for stack space, but no space for form storage. A pointer to the base of the evaluation stack is placed in the base segment.

When a stack is created a segment of minimum size is allocated to hold it. This segment contains the stack pointer. If the stack outgrows this segment another segment is requested and chained to the first. The pointer is understood to point relative to the origin of the segment which currently contains the top of the stack. When the pointer comes back down out of a stack segment, that segment is released, and the pointer is set to the top of the previous segment. The segments of a stack form a doubly-linked chain.

Forms are stored one per segment; large pieces of text must be segmented to stay within the maximum segment size restriction placed on processor space requests. The form store is a singly-linked chain of form segments. In addition to the segment extent and chain pointer, a form segment contains the length of the name string, the name string itself, and the value string. New forms are added at the beginning of the form store chain. When space must be released the form at the end of the chain is selected for writing out to disk. Each time a value is assigned to a name a new form is created; the previous instance of that name will usually be deleted. This is done even if the new value would fit into the old form segment. With forms varying in length so dynamically, it does not appear worthwhile to try to re-use an old form segment. Further, the policy of creating a new form at each assignment means that the most recently assigned form is at the head of the list. To locate a particular form by name a linear search is performed on the list. This should perform better than the average for randomly-distributed forms, since the pattern of accesses to operands during program execution is not at all random. The name search process is related to the memory management process. It is just those items which should be at the head of the list that should have priority for remaining in fast storage. If a frequently-referenced item does happen to get written out to disk it will be brought back to the head of the list at the next reference, where it will enjoy quick access for a time.

SECONDARY STORAGE

The secondary storage subsystem consists of a control processor and a rotating memory. Commands to the processor specify the main storage address of a segment to be written out, or the disk address of a segment to be read in. Writing takes precedence over reading, since it tends to free space in main storage. A write operation returns the disk address at which the segment has been stored to the user-evaluator stack; a read command returns the main storage address at which the requested segment has been loaded.

The minimum addressable amount of disk space is called a sector. Short segments will be written on disk one per sector, while longer ones will occupy several successive sectors. In a write operation the control processor must locate the first available space on the disk having the requisite number of free sectors. For this purpose a free-sector map containing one bit per sector is maintained. The bits of the map are organized into shift registers, one per disk track, which are shifted synchronously with disk rotation. The bits in the register corresponding to sectors which are approaching the disk write heads are shifted through discrete flip-flops. Simple gating of their outputs indicates the number of contiguous free sectors which can be written next. Knowing how many contiguous sectors are needed, the control processor scans over the shift registers until it has found sufficient space. It then sets the flip-flops to mark the chosen sectors in-use, performs the write, returns the disk address to the evaluator stack, and requests the allocator to release the main storage segment that has been written. Should the control processor fail to find enough space on any track it must wait until another sector time has gone by, at which time space might be sufficient. Meanwhile it can attempt to process other write requests.

Read requests are stored by the disk control processor in a list that is kept sorted by disk address. The disk sector counter is compared with the list to determine whether a read can be executed at the next sector time. If so, the processor first requests enough main storage to receive the data to be read. If this succeeds it performs the read, marks the disk sectors free in the map, and returns the main storage address to the user-evaluator stack. If the allocator fails to make the requested main storage available soon enough the read request is returned to the sorted list for a later attempt.

ACKNOWLEDGMENTS

The ideas used in this design have come from many sources. Some of the most helpful sources were Alan Kay,[8] and Rex Rice.[9]

REFERENCES

1 J HAYNES
Designing a computer: The eclectic information processing system
National Technical Information Service Springfield Virginia No N71-19923

2 C N MOOERS
TRAC—A procedure describing language for the reactive typewriter
Communications ACM Vol 9 No 3 pp 215-219 March 1966

3 C STRACHEY
A general purpose macrogenerator
Computer Journal Vol 8 No 3 pp 225-241 October 1965

4 R S BARTON
Ideas for computer systems organization: A personal survey
Software Engineering COINS III Proceedings of the Third Symposium on Computer and Information Sciences Miami Beach Florida pp 7-13 December 1969

5 D E KNUTH
The art of computer programming
Vol 1 Addison-Wesley Palo Alto 1968 pp 435-451

6 W M MCKEEMAN J T HORNING D B WORTMAN
A compiler generator
Prentice-Hall New Jersey 1970

7 P WEGNER
An introduction to stack compilation techniques
Introduction to System Programming Academic Press 1964

8 A KAY
The reactive engine
PhD Dissertation University of Utah 1969

9 R RICE et al
Papers on the SYMBOL system
AFIPS Conference Proceedings 1971 Spring Joint Computer Conference Vol 38 AFIPS Press Montvale New Jersey 1971 pp 563-616

Microtext—The design of a microprogrammed finite state search machine for full-text retrieval

by R. H. BULLEN, JR. and J. K. MILLEN

The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

The Microtext system represents a new approach to the design and implementation of a full-text retrieval system. The approach is unusual in that it integrates hardware, firmware, and software components in an attempt to provide a solution to the problems involved in processing large files of unformatted textual data. The system is based on a minicomputer specialized for high-speed full-text retrieval, through the use of a finite state search algorithm implemented in firmware.

Full-text retrieval

Full-text retrieval, as distinguished from other types of text and data processing, involves the location of patterns of characters, words, and phrases in text. In addition, bibliographic structures, such as title or author, as well as linguistic structures, such as sentence and paragraph, can be identified in text when the data base has been suitably constructed.

A variety of systems have been built to perform full-text retrieval.[1,2] If generalizations are possible, these systems can be divided into two categories:

(1) Those systems which use an index, or concordance, of text words during retrieval, but which have access to the full text for display. In some cases, such systems can also perform a sequential search of the full text for query items not also in the index. Generally speaking, search performance with indexed systems is adequate, but index generation and update is time-consuming and, as a result, editing or augmenting the full text must usually be done off-line, if at all.

(2) Those systems which always make a direct search of the full text. Such systems lend themselves well to dynamically changing file collections, because no indexing need be done; but file size is usually restricted because search time is proportional to the amount of text searched. However, because searching can be done on a character-by-character basis, direct searching can permit considerably more detailed query patterns than are possible with most indexing schemes. In addition, editing and augmentation of the full text can be performed on-line, although sometimes with side-effects which can adversely affect later search performance (e.g., file fragmentation).

Regardless of which of the above categories a full-text system may fall into, it is at an immediate disadvantage with respect to retrieval performance when compared, for example, with structured data retrieval systems (i.e., data management or management information systems). In the latter case, requests for qualifying data base entries can be satisfied by inspection of a selected subset of fields in each data base entry, whereas a full-text system must concern itself with all of the text in each entry. This problem is particularly acute for direct search full-text systems, since all of the text must be scanned *each time* a search is performed rather than only once, at index generation time, in the case of indexed systems.

Full-text systems, and specifically direct search systems, are plagued with a second problem, which is at the heart of the motivation for the Microtext system. In addition to having to process a very large amount of data in response to retrieval requests, direct search systems have a performance disadvantage because of the inability to express full-text handling functions in the primitives and data structures available on most general-purpose computers. Software must be used to map these application-level functions, often with great difficulty, into the facilities of fundamentally word- and arithmetic-oriented central processors; it is this mismatch of problem and tool, and the additional level of mapping required, which adversely affects full-text handling systems, and it is to this mismatch that the Microtext work is addressed.

An application architecture

One approach to the solution to these problems, and the approach which was taken in the development of Microtext, is to work toward the design of a computer system specialized for full-text processing and retrieval functions. The system envisioned would be built up from hardware, firmware, and software components in the following way:

(1) hardware: state-of-the-art, commercially available hardware would be used to provide a low-cost, easily reproduced base for the system. The hardware would be chosen with a view toward its eventual use by judging its inherent suitability for character string handling, its raw performance, and its ease of microprogramming.
(2) firmware: microcode would define the data structures, primitives, and basic architecture (execution environment) for text handling problems at a level which facilitates their expression. In addition, because Microtext is viewed as an application-oriented machine, many functions typically thought of as the province of an operating system would be implemented directly in microcode.*
(3) software: software would be used for most data- and user-oriented functions so that they could be easily changed to suit specific application requirements.

The question is: how to get there from here?

The Microtext develonment plan

Aside from the fact that a task of this magnitude would take considerable time and money, with few intermediate products along the way, there is also a fundamental technical problem involved here. If a designer were to take the theoretical approach and begin his task by specifying the system architecture, he might risk bounding the problem before it was identified in a practical sense. If, as in the case of Microtext, this involved a higher-level application machine specification, later changes to the system due to practical requirements could affect the basic architecture of the syst em, and changes at that late date might not be tolerable. A more conservative, practical approach was chosen for Microtext.

The development plan for Microtext makes use of a phased, or boot-strap, technique, wherein the output of each phase is an operational prototype, the application of which can proceed in parallel with the design of the following phase. The approach has the advantage that each phase can bind to the basic architecture of the system only that subset of the application environment which has been proven through practical experience and user feedback, leaving still experimental components untouched in software where they can be changed easily if the need arises.

Phase I

To test the design philosophy described above, it was decided that the first phase in the development activity should be a prototype software implementation of a full-text retrieval system, with an important, but manageable, subcomponent of the system implemented in firmware. The idea was to take a cautious, initial step, to prove the feasibility of the approach as well as to encourage, through the production of software support, the development and use of data structures and primitive operations fundamental to full-text handling problems, so that these facilities might be well enough understood to be applied in subsequent phases of the Microtext activity.

In line with this goal, a fundamental primitive of full-text retrieval—character string searching—was selected for implementation as the function of a microprogrammed, black-box peripheral device attached to a larger host computer system, specifically an IBM System 370/155. A search algorithm was designed, using techniques of finite state automata theory, and was implemented in firmware on a Digital Scientific Corporation Meta-4 computer. Higher-level language software was used to implement the control logic for the device, as well as the application logic necessary to demonstrate the operation of the system.

The sections which follow describe the overall structure of the system, the special finite state search algorithm designed for the application, and the implementation of the algorithm in firmware. A final section dis-

* A recent project at MITRE has demonstrated ways in which operating system functions can be distributed between firmware and software.[3]

cusses some of the refinements planned for the Phase I system and suggests possible directions for Phase II activity.

SYSTEM STRUCTURE

Application environment

The goal in the specification of an application environment for the initial version of the Microtext system was to model the operational characteristics of a full-text retrieval system, without actually implementing all the bells and whistles which a demanding user might desire. For this first pass, we were most interested in basic structure, not so much in form.

From the user's point of view, Microtext provides an on-line full-text retrieval capability, available through a time-sharing system* on IBM 2260 displays, 2741 terminals, and teletypes. The heart of the terminal environment is provided by three commands, described below.

DQUERY—Display query questionnaire

This command causes display at the user's terminal of a questionnaire which is used to specify basic parameter data for the search, as well as the query itself. Also specified in the questionnaire is information about the structure and format of the file to be searched.

The query language used to specify retrieval requests allows searches for words, phrases, or expressions involving words and phrases, optionally restricted in scope to the level of sentence or paragraph. This language is described in more detail in the following section.

SEARCH—Search file

This command causes the query to be processed and the search to be initiated. The query is redisplayed at the terminal for verification, and as the search progresses the search monitor displays continuous hit data to reassure the user that the system is actually working. The user controls the frequency of this output by the parameters specified in the query questionnaire. At the end of the search, the system displays the total number of documents searched as well as the number of hits.

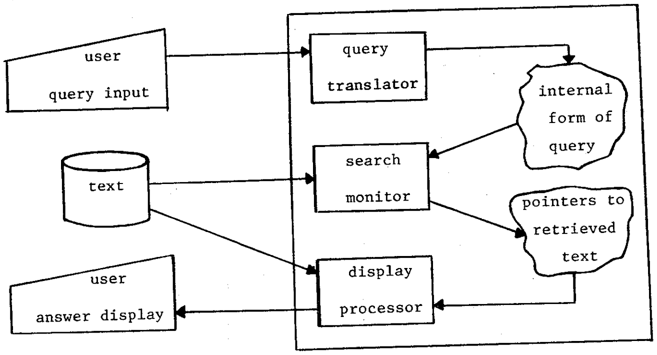


Figure 1—Flow diagram of a hypothetical full-text retrieval system

DANSWER—Display answers

This command creates a file of retrieved text and allows the user to browse through this file.

System operation

In order to understand the role of the microprogrammed processor in the Microtext system, consider first the operation of a hypothetical full-text retrieval system, as it might be driven by the set of commands described above. For this purpose, the system can be thought of as three primary modules: (1) a query translator, (2) a search monitor, and (3) a display processor. From the user's point of view, the first two of these modules are not really thought of as separate components, and in the online terminal environment described above, they are lumped together under the single "SEARCH" command. Figure 1 gives a flow diagram of such a system.

User input is first validated by the query translator and is then translated into an internal form, which facilitates easy evaluation of subparts of the query. This internal representation and the text file are then input to the search monitor which produces, not documents, but pointers to text items matched during the search. This list of pointers, and the text file, are then input to the display processor which gives the user access to the retrieved items.

It is in support of the operation of the search monitor that it was decided to apply firmware components first. To see how this was done, let us break the search monitor down further into the following processing functions:

(1) a data base interface, which is concerned with data access requirements and with the specifics

*The system under which the Microtext software runs is OS/MVT with the Time Sharing Option (TSO).

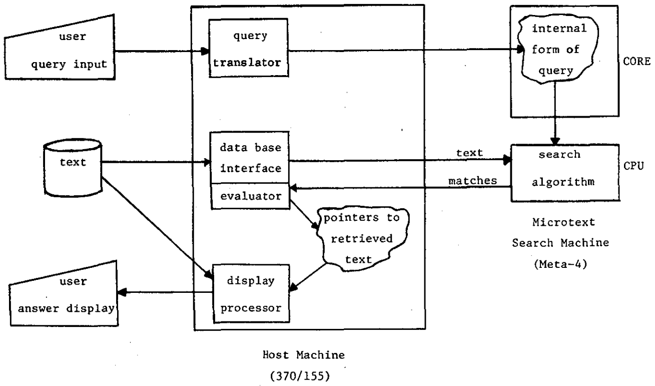


Figure 2—Retrieval operation with Microtext search machine

of file structure, and which has the function of preparing text blocks for searching;

(2) an evaluator, which drives the matching process, evaluates the query, and records hit information;

(3) and a character string searching algorithm, which performs the scanning and recognition involved in the retrieval process.

Figure 2 shows this same system, no longer hypothetical, redrawn to indicate how the Microtext search machine replaces the character string searching function of the search monitor. Here the three major components perform exactly the same functions as before, but the data flow is slightly altered. The internal form of the query, described in more detail in a later section, is in a tabular form, highly compacted to fit within the available core memory on the Microtext search machine. After the table is generated, it is sent over a high-speed interface to the Microtext machine. Control is then passed to the search monitor which accesses the text as before, but appeals to special functions which communicate directly with the search machine, through operating system I/O facilities. The results of individual matches are returned to the host machine and hit information is recorded for later use by the display processor.

The reader will note from Figure 2 that there are several other ways in which the operation of a full-text retrieval system might be shared between a host machine and a specialized, microprogrammed processor. One such way might be the inclusion in the Microtext machine of the data base interface function and the incorporation in the design of a direct connection between this machine and the data base. This would have had the obvious advantage of avoiding the extra I/O transfer of text first to the host machine and then to the search machine, but would have been inconsistent with the development goals described in the Introduction to this paper. In this initial version of the Microtext system, we wanted to separate as much as possible the well-defined problem of character string searching from such functions as the data base interface, which are more likely to be sensitive to particular application requirements. The final section of this paper presents this mode of operation, as well as other alternatives, as possible directions for future work.

THE SEARCH MACHINE

Brief description

In this section we will examine the Microtext search machine more closely. It is, of course, implemented in firmware, but before we can fully appreciate this aspect of the machine, we have to understand the driving algorithm, and the manner in which the input to that algorithm is generated.

A finite state approach to character string searching satisfies the two requirements of (1) improving the performance of the sequential search, and (2) not sacrificing in any way the user's ability to state search requests that reap the benefit of having the full text available. This is accomplished by first transforming the search request into a table using a software routine. The actual search is then performed on each section of text by a very simple microprogrammed algorithm which operates on the text, the table, and a register holding the "state" of the search. A section of text is by definition that portion of text submitted to an individual execution of the search algorithm. Its length is controlled by application software, and it could range from a sentence to a complete document. The search passes through the text section from beginning to end, using each successive character to transform the state by consulting the table. At the end of the section, the output of the resulting state indicates whether or not the section satisfies the search r equest. (In cases where the search request succeeds or fails before reaching the end of the section, the search stops immediately and restarts with the next section.)

By choosing a query language abstractly equivalent to the regular expression language of Kleene, we can employ existing algorithms to construct a finite state recognizer for strings of characters satisfying the query.* At the same time, the regular expression

* We use "query" interchangeably with "search request."

language is powerful enough to support a query language at least as flexible as those designed for existing full-text systems.[1,2]

Various ways are then available for designing a table to direct the emulation, as it were, of the finite state machine. We have chosen the straightforward tactic of constructing a deterministic transition table. A nondeterministic version of a finite state machine is generally smaller and more easily found from the regular expression; a scheme for using it for string searches was suggested by Thompson.[4] A nondeterministic search method, however, was thought less suitable for microprogrammed implementation because of the greater number of core references required per character.

It should be kept in mind that, while the table format (or choice of formats) is fixed by firmware, a new finite state machine to fill in the table must be constructed for each search request, preferably quickly enough not to discourage a user waiting at an interactive terminal.

The query language

Our present query language is regular expression notation modified for the convenience of the user. The precedence of operators has been changed to reduce the number of parentheses required in natural formulations of common search requests, and a number of standard abbreviations have been set up.

The following samples illustrate both the flavor of the present query language and the power of search requests based on regular expressions.

```
Query 1:  / MICROPROGR/  & ¬/  EMUL/
Query 2:  /(/###⟨ U  S⟩  TROOP/  & /
            ( WITHDR |  PULL- OUT)/ &
            'SENTENCE')/
```

Query 1 specifies a section of text about microprogramming but not emulation. The slashes indicate the embedding of the adjacent expressions in arbitrary text, and the blanks in contact with letters denote required punctuation or blanks. Thus, more literally, a section satisfies Query 1 if it contains a word beginning with "MICROPROG" but no word beginning with "EMUL."

Query 2 specifies a section mentioning the withdrawal of at least 100 U.S. troops. The number sign # stands for an arbitrary digit; the vertical bar is the "or" operator; the hyphen permits an arbitrary string of letters; and the angle brackets ⟨ ⟩ enclose an optional expression. In order to ensure that the "TROOP" mention is logically related to the "WITHDR" mention, they are required to be in the same sentence.

The queries are recognizable as regular expressions after the abbreviations have been expanded. For example, the slash / is translated to $\neg\phi$.* The 'SENTENCE' in quotes is an abbreviation for a moderately complicated regular expression characterizing the set of strings which can be sentences in the given data base. The option brackets are expanded so that ⟨expression⟩ becomes an "or" between the null string and "expression" (the right bracket just becomes a right parenthesis). Incidentally, the digit sign # is not expanded into (0 | . . . | 9), but is retained by the software as a single character-range symbol until the final construction of the table.

Query translation

Construction of the table from the query can be summarized in four steps:

(1) Expansion of abbreviations
(2) Infix-to-prefix translation
(3) Production of the state graph
(4) Table generation.

The Microtext implementation of this process is unusual in two ways: string manipulation techniques were used throughout (to simplify working space management and to anticipate the development of Phase II primitives), and several well-known algorithms were used in straightforward ways.

Expansion of abbreviations

While selecting the abbreviations requires some ingenuity, their expansion in the query is a simple table lookup. This is fortunate, because new abbreviations generally have to be designed for different data bases. For example, the fact that a data base may or may not have lower case letters affects the abbreviation for "arbitrary letter". Eventually Microtext software will have an associated data base descriptor file which will be used for, among other things, selecting the correct expansions. This will allow the user to express his queries in the same language, regardless of the structure of the data base he is searching.

* Regular expressions are built up from the character set using operators and two special symbols: phi (ϕ) and lambda or nil (λ). The symbol ϕ represents the empty set and $\neg\phi$, therefore, represents the set of all strings. The symbol λ represents the null string.

Infix-to-prefix translation

The regular expression resulting from expanding the abbreviations is translated from its infix-operator form to a prefix form which is both more compact and easier to manipulate symbolically. Unions, intersections, and concatenations have any number of arguments and are thus parenthesized; complements and Kleene closures (stars) have one argument and have no bounding parentheses. Zero and one are used for ϕ and λ, respectively. A sample prefix regular expression is

$$(\&{*}(|\,AB)(.\neg\,0B))$$

The infix-to-prefix translation is an instance of the classical use of a pushdown stack for this purpose. A transduction grammar of sixteen productions (exclusive of the replacement of the character set by a single nonterminal) was found and used in a simple syntax-directed translation using Lewis and Stearns' three stack algorithm.[5]

Production of the state graph

Brzozowski's derivative method was implemented to produce the state graph.[6] There is essentially only one other kind of method, based on Kleene's original proof that regular expressions can be recognized by finite state machines. It has two steps: generating a nondeterministic machine, and then converting it to a deterministic one. While this two-step method is fine in a batch system, such as the RWORD system for producing lexical processors, where it is followed by a state reduction phase, our early experiments in this direction were discouraging in speed of operation.[7]

A number of far-reaching design choices were made here for reasons of efficiency.* For example, the derivative algorithm generates a regular expression for each state, and these must be compared with previously generated ones and stored if they are new. Since even reasonable queries can give rise to large numbers of states, most state expressions are stored on disk, while a few are kept in a buffer according to a usage-age rule. Details of this and other strategies of the state graph production procedure constitute a paper in preparation.

The state graph is produced in the form of a list of transitions from each state. A transition comprises (1) an input character, (2) the next state after reading that input character, and (3) the next state output. The next state output is an indication of whether the text starting from the beginning of the section and ending with the current input character is recognized as satisfying the query. Before production of the state graph, the query is augmented slightly so that only complete sections satisfying the query are accepted by the finite state machine.

To cut down on the length of the list of transitions, certain characters are distinguished as *significant* for each state; the others share a default transition. In most states, only a few characters will be significant.

Table generation

The idea of distinguishing significant from default characters carries over into the design of the table used by the microprogrammed search algorithm. To explain the design of the table, let us shift our time frame from the preprocessing of the query to the execution of the search algorithm. During the search, the transition for the current character must be located among the set of transitions from the current state. This is done, in the table format described below, with a binary search among the significant characters with respect to the eight-bit unsigned value of the current character. Failure of this search causes the default transition to be taken.

For example, suppose state 1 has a transition to itself when the input is any letter, to state 2 on a blank, and to state 3 on any other input character. This portion of a hypothetical state graph is shown in Figure 3(a). The list of transitions from state 1 is shown in Figure 3(b). The table generator identifies the intervals of characters (considered as eight-bit unsigned binary numbers) causing each transition. It produces a list like the one in Figure 3(c). The isolated characters in the list in Figure 3(c) are the ones against which the input character is compared in the binary search for the proper transition. The search tree is shown in Figure 3(d). The order in which the comparisons are made is chosen by applying an easy modification of Huffman's algorithm to minimize the average search time (under the simplifying assumption that the successive characters in the text were chosen randomly and independently with given probabilities).[8] The probabilities can be assigned proportionally to the relative frequencies of the individual characters in a representative sample of the data base, or the usual single-letter English probabilities can be used. This procedure, while not guaranteed optimal, should result in generally better performance than, say, choosing an arbitrary balanced tree on the same characters.

* Queries like those discussed in this section have been routinely processed and generate machines of about 50 states and 300 transitions. State graph production occurs at a rate of about 20 transitions per second of CPU time.

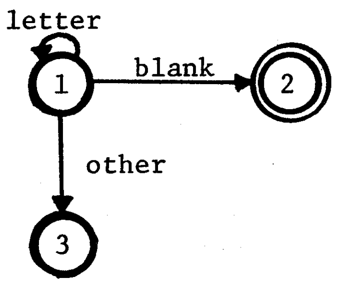


Figure 3(a). The locality of state 1 in a hypothesized state graph.

FROM STATE	ON INPUT	TO STATE	WITH OUTPUT
1	letter	1	0
1	blank	2	1
1	other	3	0

Figure 3(b). The transitions from state 1.

FROM STATE	ON INPUT	TO STATE
1	[x'00',blank)	3
1	blank	2
1	(blank,A)	3
1	A	1
1	(A,Z)	1
1	Z	1
1	(Z,x'FF']	3

Figure 3(c). The transitions by EBCDIC character value. Intervals exclude the endpoints except at x'00' and x'FF'.

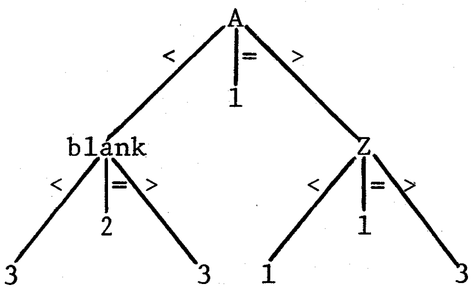


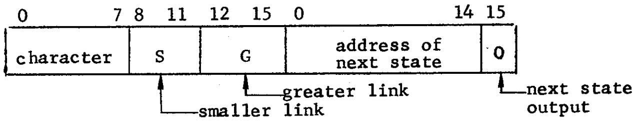
Figure 3(d). The search tree for state 1.

CHARACTER	DISPLACEMENTS SMALLER LINK	 GREATER LINK	ADDRESS OF TABLE SECTION FOR NEXT STATE	NEXT STATE OUTPUT
A	1	2	addr(1)	0
blank	2	2	addr(2)	1
Z	0	1	addr(1)	0
-	0	0	addr(3)	0

Figure 3(a)—The locality of state 1 in a hypothesized state graph
Figure 3(b)—The transitions from state 1
Figure 3(c)—The transitions by EBCDIC character value Intervals exclude the endpoints except at X'00' and X'FF'
Figure 3(d)—The search tree for state 1
Figure 3(e)—The layout of the table section for state 1

The table section for a given state can be written directly from the binary search tree. Figure 3(e) shows the layout of the section of the table for the sample state discussed above. A binary search using computed addresses, although simple in concept, is a complicated algorithm by microprogramming standards; instead, to simplify next-address calculation, the displacements from each table entry to entries for greater- and smaller-valued characters are found in the table for each transition. The table is laid out so that all of the displacements are positive. A zero displacement forces the present transition. Thus, the default transition is signalled by zeros in both link fields. Also shown in Figure 3(e) is a special feature of this choice of table format: the zero in the smaller link field for the character "Z" indicates that not only is state 1 to be the next state for the letter "Z", but also for all letters smaller than "Z". After generating all the table sections for the states in the state graph, and therefore knowing the relative addresses of the table sections for each state, the table generator makes a final pass over the entire table replacing state numbers with addresses of the corresponding table sections. Finally, the table contains, for each transition, the output associated with the next state; the use of this field is described with the search algorithm below.

The internal format of a table entry, with bit addresses for each field, is shown below.



Note that a table entry requires two 16-bit words, and that therefore the address of a table entry is always even. Thus, the last bit of the next state address is always zero, permitting the last bit position to store the output for the next state.

The search algorithm

Figure 4 shows a flowchart of the firmware search algorithm which accesses the above table format. The algorithm has three inputs, as previously mentioned: the text section, the table, and the 16-bit current state value, which is maintained in a microregister during the search. The operation of the algorithm is quite straightforward. Note that the algorithm can terminate in either of two ways: (1) if the input is exhausted before a state is encountered with an output of one, or (2) if a state with an output of one is encountered first. Although, in the state graph, the output is one only at the end of a complete section, the list of transitions is inspected before table generation for states which, once entered, cannot be left until the end of the section. Their outputs are set to one so that the search will stop there and the remainder of the section can be skipped.

Machine architecture

The architecture of the Microtext search machine is shown in Figure 5. Some basic statistics about the size of the machine are indicated in that figure; the microprogram occupies 243 microinstructions, of which approximately 20 percent are for the search algorithm itself, the rest being required for system and interface transfer control.

The device is initialized by writing the table and the initial state into the machine's core memory. The search command is then sent to the Meta-4. The search logic loads the initial state into a local store register where it remains for the duration of the search, and the search begins. As each character is accepted from the 370/155,

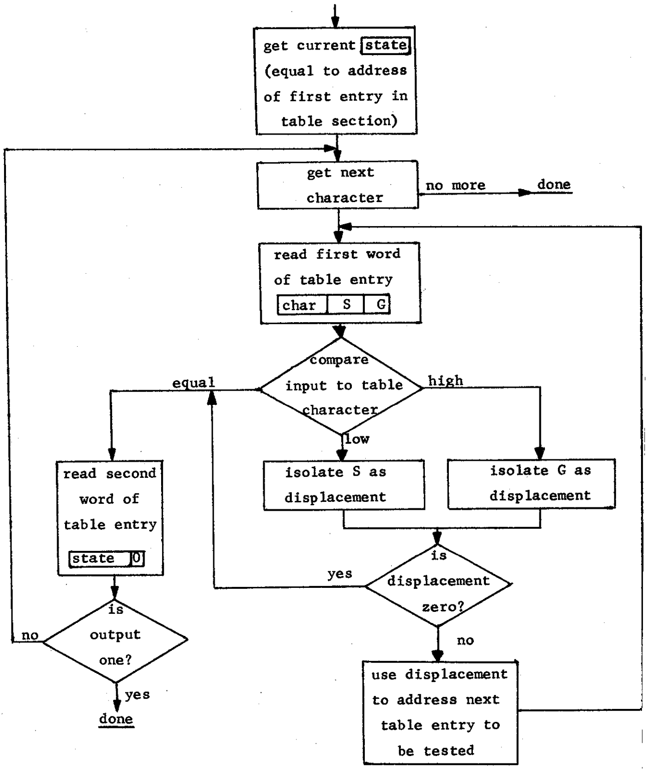

get current state (equal to address of first entry in table section)
get next character
no more
done
read first word of table entry
char S G
compare input to table character
equal
high
low
read second word of table entry
state 0
isolate S as displacement
isolate G as displacement
is displacement zero?
yes
no
is output one?
no
yes
done
use displacement to address next table entry to be tested

Figure 4—The Microtext search algorithm

the count is incremented and a transition is taken by lookup in the table in core memory. As each new state replaces the current state in local store, it is inspected to see if its output is one. If so, the search terminates immediately by presentation of ending status to the 370/155 channel. Otherwise, the search proceeds until the channel stops sending data. When termination occurs, the search logic stores the updated state and count in core memory and the control logic takes over. On the 370/155, the channel then reads back the state and count from the machine's core memory, software records hit information if a match occurred, and the search is continued with the next block of text.

Performance data

It is instructive to compare the performance of a highly specialized, microcoded algorithm of this kind to a similar approach implemented in software. In this case the most appropriate comparison for the Microtext search algorithm would be to a machine language implementation of the algorithm for representative System/360 and System/370 machines, where the search would be performed on text in a buffer in the machine's main memory. The table below compares, for each method, the minimum time in microseconds to process a single character (that is, the case where the character matches the first table entry inspected by the algorithm), and the additional time in microseconds required for each subsequent probe in the table, if previous probes do not result in a match or default condition.

	minimum time	probe time
Microcode:		
Meta-4	4.5	.9
Software:		
370/155	8.9	8.9
370/145	19.6	19.3
360/50	46.8	47.5

From the figures, it can be seen that the microcoded implementation averages several times faster than the software implementation on the 370/155. It should be noted that the large difference between minimum and probe times for the Meta-4 is due to the overhead for I/O interface transfer; the software implementations need only perform an Insert Character instruction.

At its best, the Meta-4 microprogrammed implementation can scan text at roughly 220,000 characters per second, with the rate degrading to roughly 140,000 characters per second when three additional table probes are required to locate the character being processed. The lower rate is still five times faster than software on the 370/155 and 25 times faster than the 360/50 for the same case.

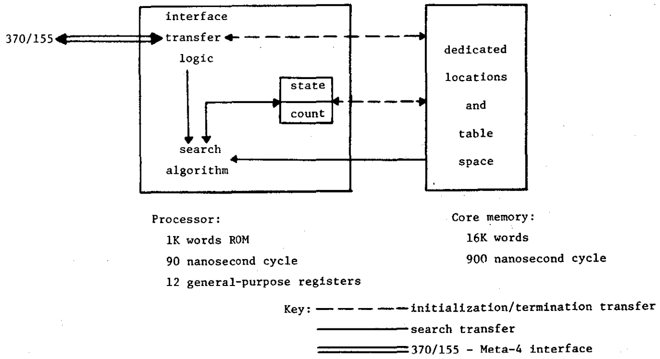


Figure 5—Architecture of the Microtext search machine

FUTURE PLANS

Future plans for the Microtext activity include completion of and refinements to the Phase I system, as well as analysis of the operation and application of the Phase I system as part of the design work for the next Phase.

Completion of Phase I

The goal of this activity is to bring the system to full operational standing, where its development can be frozen, and emphasis can be placed on its use and application. Extensions to the current software support are planned to make the system easier to use, and to improve the performance and capacity of the system in the translation of very complex queries. The microcoded search algorithm has remained stable since its implementation and no further changes to it are planned.

Planning for Phase II

We feel very strongly that the development of application-oriented systems should proceed in parallel with the application of initial, or prototype, versions of such systems. It is only through experience in the solution of real problems, and through user feedback, that truly useful automated systems can be developed. Of necessity, then, we can at best suggest possible future directions for Microtext, with specific plans waiting until we have had the benefit of this application experience.

One possible future direction is toward a version of the system which would take over query translation as well as searching responsibilities from application software. The user query would be sent directly to the Microtext processor, where it would be translated into the tabular finite state machine description. The host machine would then be notified that the processor is ready and the search would begin. This version of the Microtext processor would be able to take full advantage of the experiences of developing the query translation software for the Phase I machine. We would expect that many of the basic modules in this software would become microcoded instructions in the query translation machine, with the top level of this software becoming the Phase II machine language.

A second possibility under consideration is to implement the Microtext searching capability as an adjunct to the basic control mechanism of a disk file subsystem on a general-purpose computer. This approach, which has promise for heavily I/O-bound installations, would augment the primitive sequential and keyed lookup capabilities of such devices with a facility for, say, reading only those records which match specific patterns. Rather than transfer the retrieved records directly, this extended control mechanism could perform the entire search automatically, accumulating hit data in a separate file on disk. When the search completes, the application software could then access this file as an index into the original text file searched.

CONCLUSION

This paper has described the design and implementation of a specialized microprogrammed processor which performs character string searching for full-text retrieval applications. The activity has been successful in proving the feasibility of the approach, in identifying the basic requirements of such a system, and has pointed out areas for future work. Conclusions about the utility of the system we have developed must wait until we have had the opportunity to apply the system in solution of real problems and until we have had the benefit of user feedback.

Although the Microtext project did not set out with this particular goal in mind, we feel that the success of our work to date demonstrates the utility of firmware as a tool for application system design. In recent years, microprogramming has largely been undertaken only by computer manufacturers, universities, and some research organizations, such as MITRE. Part of the reason for this is that inexpensive microprogrammable computers have not been available for experimentation, and few guidelines have been developed for the methodology of applying microprogramming in systems design. We believe that this situation is changing, and we hope that reports of practical experience such as ours with the development of Microtext will contribute to this body of knowledge.

ACKNOWLEDGMENTS

The authors wish to acknowledge the financial support of the MITRE Independent Research Program and the contributions to the design and implementation of the Microtext machine made by H. A. Bayard, E. L. Burke, G. E. DeAgazio, R. J. Fleischer, and W. L. Schiller. Additional acknowledgment for moral support and encouragement is due J. A. Clapp, E. L. Lafferty, and C. M. Sheehan.

REFERENCES

1 R S GLANTZ
SHOEBOX—A personal file handling system for textual data
AFIPS Conference Proceedings Fall Joint Computer Conference Vol 37 1970

2 W B KEHL J F HORTY C R T BACON D S MITCHELL
An information retrieval language for legal studies
Communications of the Association for Computing Machinery Vol 4 1961

3 B H LISKOV
The design of the VENUS operating system
Communications of the Association for Computing Machinery Vol 15 1972

4 K THOMPSON
Regular expression search algorithm
Communications of the Association for Computing Machinery Vol 11 1968

5 P M LEWIS II P E STEARNS
Syntax directed transduction
Journal of the Association for Computing Machinery Vol 15 1968

6 J A BRZOZOWSKI
Derivatives of regular expressions
Journal of the Association for Computing Machinery Vol 11 1964

7 W L JOHNSON J H PORTER S I ACKLEY D T ROSS
Automatic generation of efficient lexical processors using finite-state techniques
Communications of the Assiciation for Computing Machinery Vol 11 1968

8 D A HUFFMAN
A method for the construction of minimum redundancy codes
Proceedings I R E Vol 40 1952

Design of the Burroughs B1700

by W. T. WILNER

Burroughs Corporation
Goleta, California

INTRODUCTION

Procrustes was the ancient Attican malefactor who forced wayfarers to lie on an iron bed. He either stretched or cut short each person's legs to fit the bed's length. Finally, Procrustes was forced onto his own bed by Theseus.

Today the story is being reenacted. Von Neumann-derived machines are automatous malefactors who force programmers to lie on many procrustean beds. Memory cells and processor registers are rigid containers which contort data and instructions into unnatural fields. As we have painfully learned, contemporary representations of numbers introduce serious difficulties for numerical processing. Manipulation of variable-length information is excruciating. Another procrustean bed is machine instructions, which provide only a small number of elementary operations, compared to the gamut of algorithmic procedures. Although each set is universal, in that it can compute any function, the scope of applications for which each is efficient is far smaller than the scope of applications for which each is used. Configuration limits, too, restrict information processing tasks to sizes which are often inadequate. Worst of all, even when a program and its data agreeably fit a particular machine, they are confined to that machine; few, if any, other computers can process them.

In von Neumann's design for primordial EDVAC,[1] ridigity of structure was more beneficial than detrimental. It simplified expensive hardware and bought precious speed. Since then, declining hardware costs and advanced software techniques have shifted the optimum blend of rigid versus variable structures toward variability. As long ago as 1961, hardware of Burroughs B5000[2] implemented limitless main memory using variable-length segments. Operands have proceeded from single words, to bytes, to strings of four-bit digits, as on the B3500. The demand for instruction variability has increased as well. The semantics of the growing number of programming languages are not converging to a small set of primitive operations. Each new language adds to our supply of fundamental data structures and basic operations.

This shifting milieu has altered the premises from which new system designs are derived. To increase throughput on an expanding range of applications, general-purpose computers need to be adaptable more specifically to the tasks they try to perform. For example, if COBOL programs make up the daily workload, one's computer had better acquire a "Move" instruction whose function is similar to the semantics of the COBOL verb MOVE. To accommodate future applications, the variability of computer structures must increase, in yet unknown directions. Such flexibility reminds one of Proteus, the mythological god who could change his shape to that of any creature.

DESIGN OBJECTIVE

Burroughs B1700 is a protean attempt to completely vanquish procrustean structures, to give 100 percent variability, or the appearance of no inherent structure. Without inherent structure, any definable language can be efficiently used for computing. There are no word sizes or data formats—operands may be any shape or size, without loss of efficiency; there are no *a priori* instructions—machine operations may be any function, in any form, without loss of efficiency; configuration limits, while not totally removable, can be made to exist only as points of "graceful degradation" of performance; modularity may be increased, to allow miniconfigurations and supercomputers using the same components.

Design rationale

The B1700's premise is that *the effort needed to accommodate definability from instruction to instruction* *is less than the effort wasted from instruction to instruction when one system design is used* for all applications. With definable structure, information is able to be represented according to its own inherent structure. Manipulations are able to be defined according to algorithms' own inherent processes. Given such freedom, it is easy to construct novel machine designs which are 10 to 50 times more powerful than contemporary designs, and which can be interpreted by the B1700's variable-micrologic processor using less than 10 to 50 times the effort, resulting in faster running times, smaller resource demands, and lower computation costs.

GENERAL DESIGN

To accomplish definable structure, one may observe that during the next decade, something less than infinite variability is required. As long as control information and data are communicated to machines through programming languages, the variability with which machines must cope is limited to that which the languages exhibit. Therefore, it is sufficient to anticipate a unique environment for each programming language. In this context, absolute binary decks, console switches, assembly languages, etc., are included as programming language forms of communication. Let us call all such languages "S-languages" ("S" for "soft," or also for "system" or "source" or "specialized" or "simulated"). Machines which execute S-language directly are called "S-machines." The B1700's objective, consequently, is to emulate existing and future S-machines, whether these are 360's, FORTRAN machines, or whatever. Rather than pretend to be good at all applications, the B1700 strives only to interpret arbitrary S-language superbly. The burden of performing well in particular applications is shifted to specific S-machines.

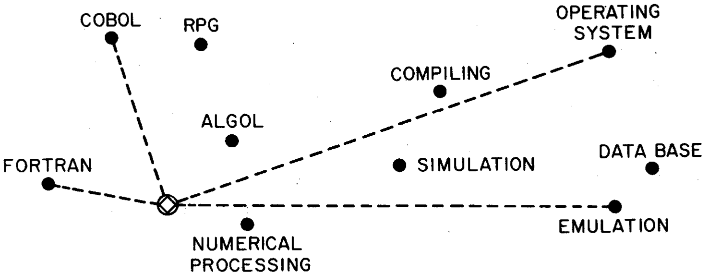


Figure 1—Typical machine design (○) positioned by goodness-of-fit to application areas (•)

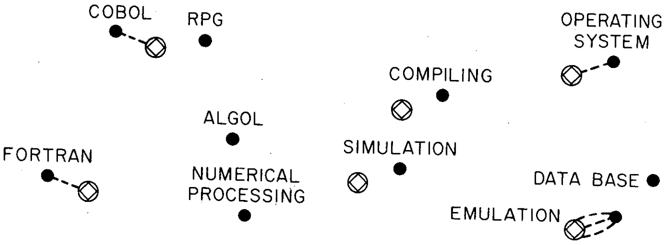


Figure 2—Typical B1700 S-machines (○) positioned by goodness-of-fit to application areas (•)

Throughput measurements, reported below, show that the tandem system of:

APPLICATION PROGRAM,
interpreted by an
S-MACHINE (which is optimized for the application area),
interpreted by the
B1700 HARDWARE (which is optimized for interpretation)

is more efficient than a single system when more than one application area is considered. It is even more efficient than conventional design for many individual application areas, such as sorting.

To visualize the architectural advantage of implementing the S-machine concept, imagine a two-dimensional continuum of machine designs, as in Figures 1 and 2. Designs which are optimally suited to specific applications are represented by bullets (•) beside the application's name. The goodness-of-fit of a particular machine design, which is represented as a point (○) in the continuum, to various applications is given by its distance from the optimum for each application; the shorter the distance, the better the fit, and the more efficient the machine is. Figure 1 dramatizes the disadvantage of using one design for COBOL, FORTRAN, Emulation, and Operating System applications. Figure 2 pictures the advantage of emulating/interpreting many S-machines, each designed for a specific application. Note that emulation inefficiencies must be counted once for each S-machine, since they are all interpreted.

HARDWARE CAPABILITIES

To allow the user's problem statement to dictate the structure of the machine and the semantics of machine operations, new degrees of flexibility and speed are required from hardware, firmware, and software.

Defined-field capability

All information in a B1700 system is represented by *fields*, which are recursively defined to be either *bit strings* or strings of fields. Specifically, bytes and words do not exist.

- All memory is addressable to the bit.
- All field lengths are expressable to the bit.
- Memory access hardware must fetch and store one or more bits from any location with equal facility. That is, there must be no penalty and no premium attached to location or length.
- All structured logic elements in the processor can be used iteratively and fractionally under microprogram control, thus effectively concealing their structure from the user. Iterative use is required for operands which contain more bits than a functional unit can hold; fractional use is required for smaller operands.

Defined-field design gives flexibility because information is represented by recursively defined structures of bits. It also gives speed because all bits in a field (and only those bits in a field) are processed in parallel. Additional speed is obtained from the advanced technology of the B1700 components. Main memory is constructed out of LSI MOS circuits with 1024-bit chips having 180-nsec access time. The B1700 is the first small-scale, general-purpose, commercial computer to use MOS/LSI circuitry in its main memory.

Generalized language interpretation

No machine language is built into the hardware. There is no processor structure or set of machine instructions for which compilers may generate code. Each language to be executed must first configure the B1700 processor into whatever structure is efficient for algorithms in that language. Defined operations on the defined structure are then executed by changeable microprogram. B1700 processors are specifically designed to avoid causing significant differences in efficiency due to differences in such "soft" machine structures and operations.

- Microinstructions are executed at 2, 4, and 6MHz rates using MSI CTL II logic with typical delay of 3 nsec per gate.
- Microcode executes out of main memory. It may be buffered through 60-nsec access bipolar circuits. Such buffering is invisible to the microprogrammer.
- Microprocedures are reentrant and recursively usable; each processor includes a 32-deep stack for fast entry and exit; stack operations are automatic, not microprogrammed.
- Microprograms are not limited in size, nor would large microprograms be inefficient because of size.
- Microcode on the B1700 is compact, economizing storage. COBOL, FORTRAN, BASIC, and RPG language processors as well as second-generation and third-generation emulators have been microprogrammed each in less than 4000 16-bit microinstructions.
- Hardware assists with the concurrent execution of many microprogrammed interpreters. It takes from 14 μsec to 53 usec (at 6MHz) from the completion of an S-instruction for one interpreter until the beginning of an S-instruction for another interpreter, depending on how much of the processor must be reconfigured.

Memory protection, fast interrupt response, and uniform status of microprograms allow each microprogrammer to be unconcerned that other interpreters may be running simultaneously.

Control over binding

While the hardware for defined-field and generalized language interpretation allows a varying processor image for microinstruction to microinstruction, it does not preclude taking advantage of a static processor image. For example, the number of bits to be read, written, or swapped between processor and memory can be different in consecutive microinstructions, but if an interpreted S-machine's memory accesses are of uniform length, this length can be factored out of the interpreter, simplifying its code. In other words, S-memory may be addressed by any convenient scheme; bit addresses are available, but not obligatory for the S-machine.

With these hardware advances, language-dependent features such as operand length are unbound inside the processor and memory buss, except during portions of selected microinstructions. Some of these features have, until now, been bound before manufacture, by machine designers. Language designers and users have been able to influence their binding only indirectly, and only on the next system to be built. On the B1700, the delayed binding of these features, delayed down to the

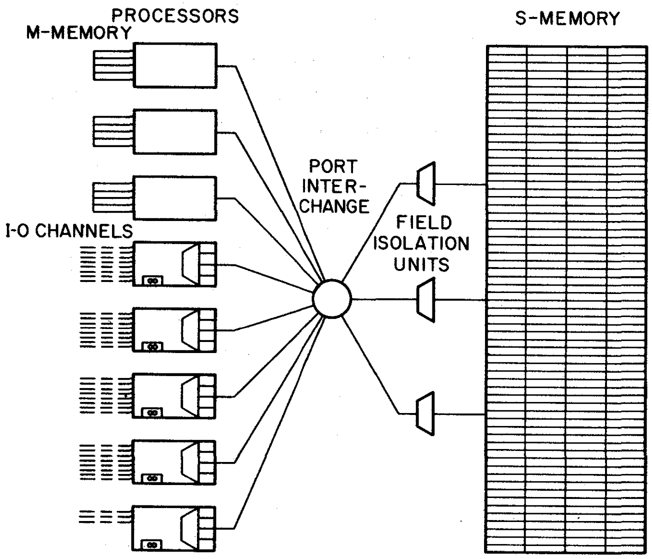


Figure 3—B1700 Organization—Peripherals include standard large-scale devices, data communications networks, and mass storage units as well as minicomputer devices such as paper tape and 96-column card equipment. Special purpose devices include graphics, document sorters, teller machines, etc.

clock pulse level of the machine, gives language designers and users a new degree of flexibility to exploit. Hopefully, this flexibility will lead to the design of languages which are levels closer to user problems. Because of the B1700's interpretation speed, there should be little execution penalty incurred by such advanced forms of man-machine communication.

SYSTEM ORGANIZATION

Extreme modularity improves the B1700's ability to adapt to an installation's requirements. There may be one to eight processors connected to one another and to two to 256 65,536-bit systems memory (*S-memory*) modules, interfaced by a field-isolation unit. ("Field-isolation" refers to converting defined-field memory requests [i.e., least- or most-significant bit address, field length, and direction] into whatever form actually drives the memory and to converting bit strings into whatever form is actually read and written by the memory.) Each processor also connects to one to eight I/O channels or to one to four microprogram memory (*M-memory*) modules. (See Figure 3.) Later systems may have several field-isolation units. With only one processor, the port interchange may be eliminated, as in Figure 4.

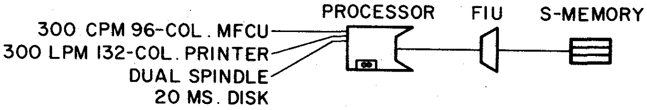


Figure 4—One of the smallest B1700's

EMULATION VEHICLE

Any computer which can handle the B1700's port-to-port message discipline may employ a B1700 for on-line emulation. (See Figure 5.) Programs and data are sent to the B1700 for execution; I/O requests are sent back to the host which uses its own peripherals for them. Interpreters are loaded via the B1700's console cassette drive. Each Burroughs emulator can run standing-alone, or in an emulation vehicle, or in a multiprogrammed mix.

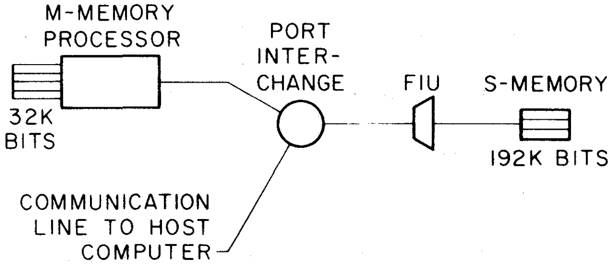


Figure 5—B1700 as an emulation vehicle

STATE OF THE ART DESIGN

The B1700's innovative features have been realized without diminishing the system's ability to provide many proven throughput enhancements. All Burroughs interpreters rely on the B1700's Master Control Program (MCP) for:

- Virtual memory—user programs are not limited in size by the amount of physical storage nor does the programmer ever need to know how much storage is available; compilers automatically segment programs, and the MCP automatically manages these segments without introducing any code into the user program.
- Multiprogramming—because common system

functions such as input/output, storage management, and peripheral assignment are removed from user programs and handled by the MCP, every pause in a running program becomes an evident opportunity to run other programs.

- Multiprocessing—with S-machine state kept in main memory and with every interpreter in main memory, any processor in the system can resume execution of an interrupted program.

The B1700 is the first small-scale computer to offer so comprehensive an operating system.

In addition to the MCP capabilities, there are notable system flexibilities, viz:

- Dynamic system configuration—processors, memory addresses, I/O channels, and peripherals are not uniquely coded into programs, so such entities can be brought on-line and used immediately without any reprogramming.
- Descriptor-organized I/O—in effect, I/O has its own S-language, interpretation of which causes data transfer; it is possible to build this interpretation in hardware, for maximum speed, or it may be soft for maximum flexibility, for example, to allow easy interfacing with new devices.
- System performance monitoring—interpreters automatically gather dynamic execution frequencies of program components to establish which parts of a program take the most time;[3,4] also, specific microinstructions can interface directly with external monitors, allowing soft event flagging.

Interpreter switching

Note that without a native machine language, the MCP itself must be written in higher-level language and interpreted just like any other program. It, and all other active jobs, are represented in memory according to Figure 6. There are read-only code segments which may be anywhere in memory and a write-protected area which contains the program's S-machine state, data segments, file buffers, and other work areas.

One of the MCP's data segments contains an interpreter dictionary that points to each interpreter which is active (i.e., interpreting one of the jobs in the mix). To reinstate a user's interpreter, the MCP extracts from the user's S-machine state the name of the interpreter being used, brings it into S-memory, and calls the interpreter interface routine which switches run structures. Associating S-machines and interpreters symbolically allows such things as several COBOL interpreters active in one mix—one designed for speed, another for code compaction, etc.—all employing the same S-language expressly designed for COBOL, that is, a COBOL-machine definition. The interpreter name is looked up in the interpreter dictionary to yield a pointer to the interpreter code in S-memory.

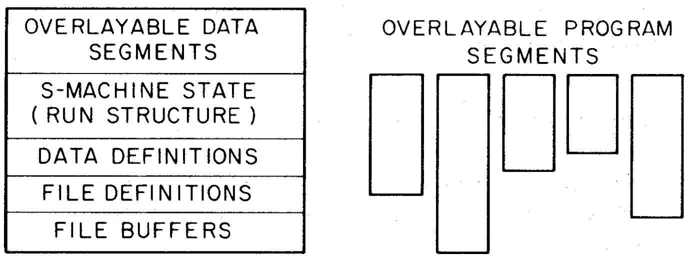

OVERLAYABLE DATA SEGMENTS
S-MACHINE STATE (RUN STRUCTURE)
DATA DEFINITIONS
FILE DEFINITIONS
FILE BUFFERS
OVERLAYABLE PROGRAM SEGMENTS

Figure 6—B1700 program S-memory components

To switch back to the MCP interpreter, a user interpreter performs the identical procedure. It calls the interpreter interface routine, which maintains a pointer to the MCP's interpreter, and switches run structures.

Interpreter switching is independent of any execution considerations. It may be performed between any two S-instructions, even without switching S-instruction streams. That is, an S-program may direct its interpreter to summon another interpreter for itself. This facility is useful for changing between tracing and non-tracing interpreters during debugging.

Interpreter switching is also independent of M-memory. Microcode always actually addresses S-memory. In case M is present, special hardware diverts fetches to it. Without M, no fetches are diverted.

Interpreter management

Entries in the interpreter dictionary are added whenever a job is initiated which requests a new interpreter. Interpreters usually reside on disk, but may be read in from tape, cards, cassettes, data comm, or other media. They have the same status in the system that object code files, source language files, data files, compiler files, and MCP files all share: symbolically-named, media-independent bit strings. While active, a copy is brought from disk, to be available in main memory for direct execution. The location may change during interpretation due to virtual S-memory management, so microinstructions are location-independent.

At each job initiation and termination, the MCP rearranges the interpreters in M-memory to try to avoid swapping. Interpreter profile statistics show that over 99 percent of all microinstructions are executed out of M-memory, even when the demand for M-memory space is double the supply. At higher demand rates, swapping occurs.

Ease of microprogramming

Writing microprograms for the B1700 is as simple, and in some ways simpler, than writing FORTRAN subroutines:

- Microprograms consist of short, imperative English-like sentences and narrative comments. For example, one microinstruction in the FORTRAN interpreter is coded as follows:

  Read 8 bits to T counting FA up and FL down.
- Knowledge of microinstruction forms is not beneficial. Although microprogrammers on other machines need to know which bits do what, on the B1700, there is no way to use that information. Once the function is given in English, its representation is immaterial. The B1700 microprogrammer has only one set of formats to worry about: those belonging to the S-language which he is interpreting.
- Multiprogramming of microprograms is purely an MCP function, carried out without the microprogrammer's knowledge or assistance. Actually, there is nothing one would do differently, depending on whether or not other interpreters are running simultaneously.
- Use of M-memory is purely an MCP function; users cannot move information in and out of M. Other than rearranging one's interpreter according to usage, there is nothing one should microprogram differently depending on whether microinstructions are executing out of M-memory or S-memory. Maximizing use of system resources is beyond the scope of any individual program; responsibility lies solely with the MCP and the machine designers.
- Since all references are coded symbolically, protection is easy to assure. Microprograms can reference only what they can name, and they can only name quantities belonging to themselves and their S-machines. Moreover, artificially generated names (e.g., negatively subscripted FORTRAN arrays) are checked for validity by concurrent hardware.

```
(a) ? COMPILE XCOBOL/INTERP WITH
    MIL; DATA CARD
(b) ? COMPILE XCOBOL/INTERP WITH
    MIL; MIL FILE CARD=XCOBOL/
    SOURCE
```

Figure 7—Typical MCP control information for creating interpreters

```
(a) ? EXECUTE FILE/UPDATE
(b) ? EXECUTE FILE/UPDATE; INTERP
    =XCOBOL/INTERPRETER
```

Figure 8—Typical MCP control information for executing programs

- Calling out interpreters is simplified by the continuation of Burroughs' "one-card-of-free-form-English" philosophy of job control language. Figure 7 shows the control information which creates a new interpreter (a) from cards, and (b) from a disk file named XCOBOL/SOURCE.
- Association of interpreters and S-language files occurs at run-time. Figure 8 shows the control information which executes a COBOL program named FILE/UPDATE with (a) the usual COBOL interpreter, and (b) another interpreter named XCOBOL/INTERPRETER.
- There is no limit to the number of interpreters that may be in the system (except that no more than 2^{44} bits are capable of being managed by the B1700's present virtual memory property, so a 28,000-bit average interpreter length means there is a practical limit of 628,292,362 interpreters . . . many more than the number of S-languages in the world).

Additional information about B1700 microprogramming may be found in Reference 5.

EVALUATION

Evaluation of novel architecture is not merely an unsolved problem; most rational attempts produce worse results than subjective guesses. Consider benchmarks, which measure more system parameters than any other technique. Any benchmark program which runs on the B1700 develops not only an observed running time, but also a program profile which indicates how to reduce that time (possibly by 50 percent or more). What, then, is the true performance of the system? The observed time, even though known inefficiencies are pin-pointed? Half the observed time? Not until the benchmark has been changed.

The point of benchmarks is to have a standard reference which allows the customer to characterize his work and obtain a cost/performance measure. What customer would be satisfied with an inefficient characterization? If the B1700 can show that a program is not using the system well, what good is it as a benchmark? If we change the program to remove the inefficiencies, it is no longer standard. This is a pernicious dilemma.

Even the simplest measure, add time, still published as if it hasn't been a misleading and unreliable indicator for the past 15 years, is void. What is the relative performance of two machines, one of which can do an almost infinite variety of additions and the other of which can do only one or two? The B1700 can add two 0-24 bit binary or decimal numbers in 187 nsec; how fast must a 16-bit binary machine be in order to have an equivalent add time?

Assuming reasonable benchmark figures are obtainable, they would say nothing about the intrinsic value of a machine which can execute another machine's operators, for both existing and imaginary computers; which can interpret any current and presently conceivable programming language; which can always accept one more job into the mix; which can add on one more peripheral and one more memory module, to grow with the user; which can interpret one more application-tailored S-machine; which can tell a programmer where his program is least efficient; which can continue operation in spite of failures in processing, memory, and I/O modules. These characteristics of the B1700, shared by few other machines—no machine shares them all—save time and money, but are not yet part of any performance measurement.

Despite the nullification of measures with which we are familiar and the gargantuan challenge of measuring the B1700's advancements of the state-of-the-art, there are, nevertheless, some quantifiable signs that the system gives better performance than comparably-priced and higher-priced equipment.

Utilization of memory

Defined-field design's major benefit is that information can be represented in natural containers and formats. Applied to language interpretation, defined-field architecture allows S-language definitions which are more efficient in terms of memory utilization than machine architectures which have word- or byte-oriented architecture. For example, short addresses may be encoded in short fields, and long addresses in long fields (assuming the interpreter for the language is programmed to decode the different sizes). Alternatively, address field size may be a run-time parameter determined during compilation. That is, programs with fewer than 256 variables may be encoded into an S-language that uses eight-bit data address fields. Even the fastest microcode that can be written to interpret address fields is able to use a dynamic variable to determine the size of the field to be interpreted.

Language of Sample	Aggregate Size on B1700	Aggregate Size on Other	Other System	Percent Improved B1700 Utilization
FORTRAN	280KB	560KB	System/360	50
FORTRAN	280KB	450KB	B3500	40
COBOL	450KB	1200KB	B3500	60
COBOL	450KB	1490KB	System/360	70
RPG II	150KB	310KB	System/3	50

Figure 9—Amount of program compaction on B1700

Just how efficient this makes S-languages is difficult to say because no standard exists. What criterion will tell us how well a given computer represents programs? What "standard" size does any particular program have? We would like a measure that takes a program's semantics into account, not just a statistical measure such as entropy.

If we simply ask how much memory is devoted to representing the object code for a set of programs, we find the statistics of Figure 9.

In short, the B1700 appears to require less than half the memory needed by byte-oriented systems to represent programs. Comparisons with word-oriented systems are even more favorable.

As to memory utilization, the advantage of the B1700 is even more apparent. Consider two systems with 32KB (bytes) of main memory, one a System/3, the other a B1700. Suppose a 4KB RPG II program is running on each. If we ask how much main memory is in use, we find the comparison of Figure 10.

The utilization at any given moment may be 30 times better on the B1700 than on the System/3. At least, with all program segments in core, it is seven times better (4.5KB vs. 32KB). Even if we assume the RPG interpreter is in main memory and is not shared by other RPG jobs in the mix, the comparison varies

System	Bytes in Use	Percent	Comment
System/3	32K	100	28K is idle without multiprogramming and virtual memory.
B1700	1K	3	Assumes 500B run structure and 500B of program and data segments.

Figure 10—Hypothetical RPG memory requirements

from 6:1 to 4:1, 5KB to 8KB (vs. 32KB), 84 to 75 percent better utilization. As more and more RPG jobs become active in the mix, the effect of the interpreter diminishes, but then comparison becomes meaningless, because other low-cost systems cannot handle so large a mix. (Note that these figures change when a different main memory size is considered, so the comparison is more an illustration of the advantage of the B1700's variable-length segments and virtual memory than of its memory utilization.) More detailed information on memory utilization may be found in Reference 6.

Running time

Although program running time is said to involve less annual cost at installations than the unquantifiable parameter which we may call "ease of use", let us mention some current observations. When the B1700 interprets an RPG II program, the average S-instruction time is about 35 microseconds, compared to System/3's 6-microsecond average instruction time. On a processor-limited application (specifically, calculating prime numbers), the identical RPG program runs in 25 seconds on a B1700 and 208 seconds on a System/3 model 10. Both systems had enough main memory to contain the complete program; only the memory and processor were used.

The B1700 lease rate was 75 percent greater than the System/3's. In terms of cost, the B1700 run consumed 30¢ while the System/3 run took $1.60. In terms of instruction executions, the B1700 was 50 times faster. That is, each individual interpreted RPG instruction, on the average, contributed as much to the final solution as 50 System/3 machine instructions. The fact that the B1700's S-machine for RPG is 50 times more efficient than System/3 seems to support the B1700 philosophy, that interpretation of S-machines which are optimized for each application yields better performance than using a general-purpose architecture.

Using another set of benchmark programs (for banking applications), and another B1700 which leases for the same as the System/3 with which it was compared, throughput comparisons are again noteworthy. Despite defined-field design, soft-interpretation, soft I/O, multiprogramming, multiprocessing, and virtual memory, all of which supposedly trade speed for flexibility, the B1700 executes RPG programs in 50 to 75 percent of the System/3 time, and compiles them in 110 percent of the System/3 time, for the same monthly rental. In applications of this type, compilation is expected annually (monthly at worst) while execution is expected daily. (Systems used for this comparison included a multi-function card unit to read, print, and punch 96-column cards, a 132-position 300 lpm printer, a dual spindle 4400 bpi disk cartridge drive, and operator keyboard. The System/3 could read cards at 500 cpm, while the B1700 could read at 300 cpm.)

CONCLUSION

Microprogramming, firmware, user-defined operators, and special-purpose minicomputers are being touted as effective ways to increase throughput on specific applications while decreasing hardware costs. One standard system tailors itself to an installation's needs. Effective as these approaches are, they are all held back by procrustean machine architecture. Burroughs B1700 appears to eliminate inherent structure by its defined-field and soft interpretation implementation, advancements of the state-of-the-art. Without a native machine language, the B1700 can execute every machine language well, eliminating nearly all conversion costs. Designed for language interpretation rather than general-purpose execution, the B1700 can run every programming language well, reducing problem-solving time and expense. It does not waste time or memory overcoming its own physical characteristics; it works directly on the problems. Furthermore, these innovations are available in low-cost systems that yield better price/performance ratios than conventional machinery.

ACKNOWLEDGMENT

Many of the design objectives were first articulated by R. S. Barton.[7] The author wishes to thank Brian Randell, R. R. Johnson, Rod Bunker, Dean Earnest and Harvey Bingham for their conscientious criticism of various drafts of this article.

BIBLIOGRAPHY

1 A W BURKS H H GOLDSTINE
J VON NEUMANN
Preliminary discussion of the logical design of an electronic computing instrument
A H TAUB (ed) *Collected Works of John von Neumann* Vol 5
The Macmillan Co New York 1963 pp 34-79
Also in
C G BELL A NEWELL
Computer structures: Readings and examples
McGraw-Hill Book Co 1971 pp 92-119

2 W LONERGAN P KING
Design of the B5000 system
Datamation 7 5 May 1961 pp 28-32

3 S C DARDEN S B HELLER
Streamline your software development
Computer Decisions 2 10 October 1970 pp 29-33

4 D E KNUTH
An empirical study of FORTRAN programs
Software—Practice and Experience 1 2 April 1971 pp 105-134

5 W T WILNER
Microprogramming environment on the Burroughs B1700
IEEE CompCon '72
For reprints write to the author at Burroughs Corporation 6300 Hollister Avenue Goleta California 93017

6 W T WILNER
Burroughs B1700 memory utilization
Proc FJCC '72 this volume

7 R S BARTON
Ideas for computer systems organization: A personal survey
Software Engineering 1 Academic Press New York 1970 pp 7-16

An on-line two-dimensional computation system*

by THOMAS G. WILLIAMS

System Development Corporation
Santa Monica, California

INTRODUCTION

The role of graphics in interactive man-computer systems is to extend the capability of the computer for communication in a visual mode so that men can communicate with the computer directly in the figurative notations and graphics conventions that they have developed for communication among themselves. A large number of computer-graphics systems have been developed, most of them directed toward drawing lines, curves, or shapes, as in schematic drawing,[1] solid and half-tone drawing,[2] and computer animation.[3] Considerable work has also been done in computer output of drawings and graphs.

This paper presents the results of an exploration into a different domain of computer graphics—one in which symbols and alphanumeric characters are the primary notations, rather than lines or pictures. Such notations are used in mathematics, organic chemistry, flowcharting, and other applications. By design and evolution, they tend to exhibit the structure, organization, and nature of the problems they are designed for in a more compact and economical manner than do notation systems (e.g., programming languages) designed to express the operational steps of logic-oriented computer programs. Moreover, these notations are familiar, through education and experience, to a greater number of potential computer users than are programming languages.

The experimental system described here, called The Assistant Mathematician (TAM), uses computer-graphics techniques to allow the on-line use of ordinary hand-printed mathematical notation for computer programming and mathematical problem solving. The second section gives a general description of the TAM facility, with examples of its use. The third section, the main body of the paper, describes the TAM software modules and some of the details of their operation. The fourth section draws some tentative conclusions about the usefulness and feasibility of on-line computation systems and indicates some areas for future refinement.

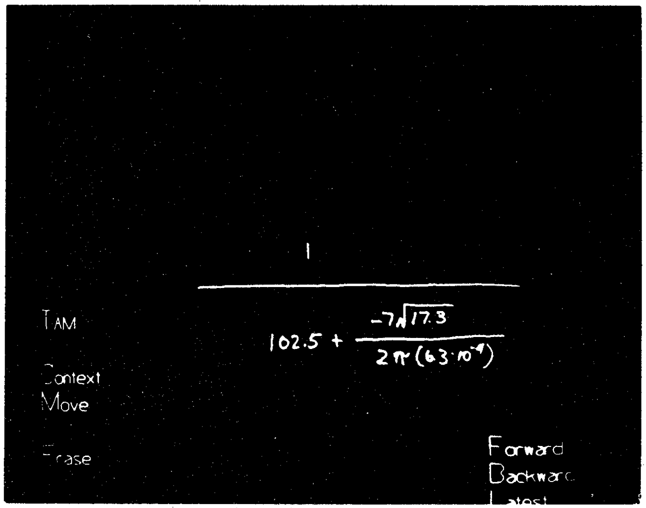
GENERAL DESCRIPTION

TAM is an interactive programming system for numeric computation. The TAM user language is ordinary two-dimensional mathematical notation. TAM incorporates an extensive set of arithmetic operators on constants, variables, and one- and two-dimensional arrays. It provides many common functions such as trigonometric and logarithmic functions. It also provides function definition and looping facilities for repetitive calculation. TAM operates under ADEPT, a time-sharing system developed at SDC.[4]

The graphics console used with TAM is a single interactive input-output surface. For input, a data tablet supplies a continuous stream of $X-Y$ coordinate pairs representing the position of the tablet stylus. Information generated by the computer program is rear-projected on the tablet surface by a cathode-ray-tube projection system. The tablet surface is the only working area on the console; no mechanical pushbuttons or keyboards are used. Printing on the tablet surface is remarkably similar to writing with a pen on a piece of paper. The engineering and interface aspects of this device have been documented by Gallenson.[5]

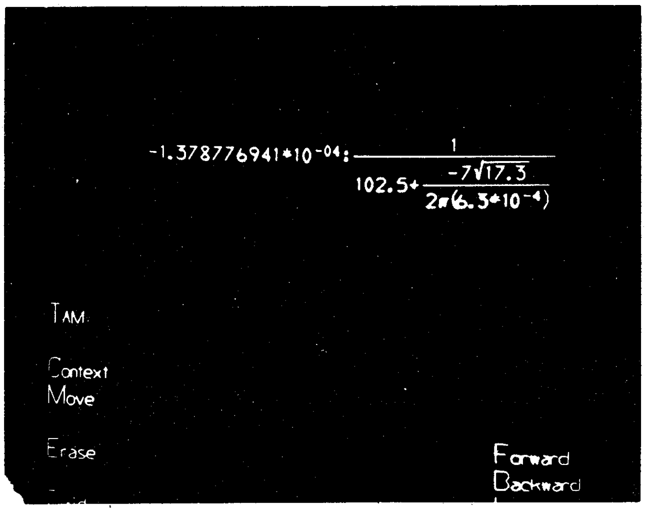
A user begins by printing the expression he wants the computer to evaluate. As he prints, the track of the stylus is displayed on the surface so that he can see what he has printed. (Figure 1 shows the console surface with hand-printed input.) When he has completed his input (signalled by a time-out), each input character is processed by a character-recognition program that operates under ADEPT. Two kinds of verification are then made. First, the character recognizer

* This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-67-C-0149.



$$\frac{1}{102.5+\frac{-7\sqrt{17.3}}{2\pi(6.3\cdot 10^{-4})}}$$

Figure 1—Handprinted input

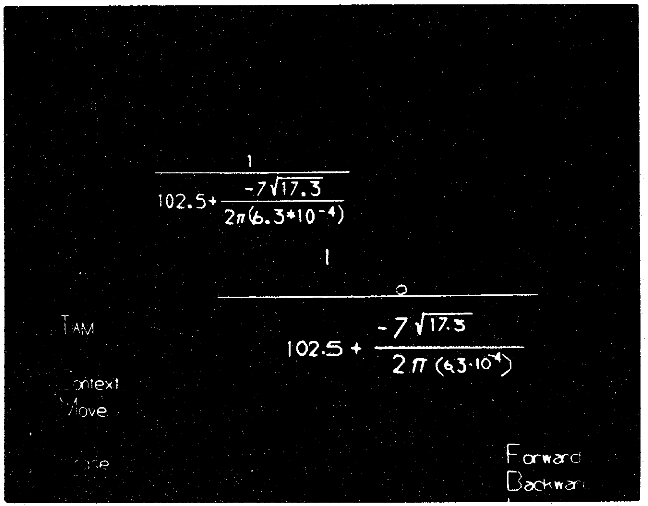


$$-1.378776941*10^{-04}:\frac{1}{102.5+\frac{-7\sqrt{17.3}}{2\pi(6.3*10^{-4})}}$$

Figure 3—Result of computation

displays a computer-generated set of characters corresponding to the position and size of the user's handprinted input; from this display, the user can verify that his input has been correctly recognized, character by character. At the same time, the character recognizer's output is passed through two mathematics-structure modules: one, the analyzer, transforms the user's two-dimensional input into a machine-interpretable linear infix statement; the other, the builder, transforms the linear statement back into two-dimensional form for display to the user, thus giving him the opportunity to verify that the analyzer is correctly analyzing his input. Figure 2 illustrates these operations.

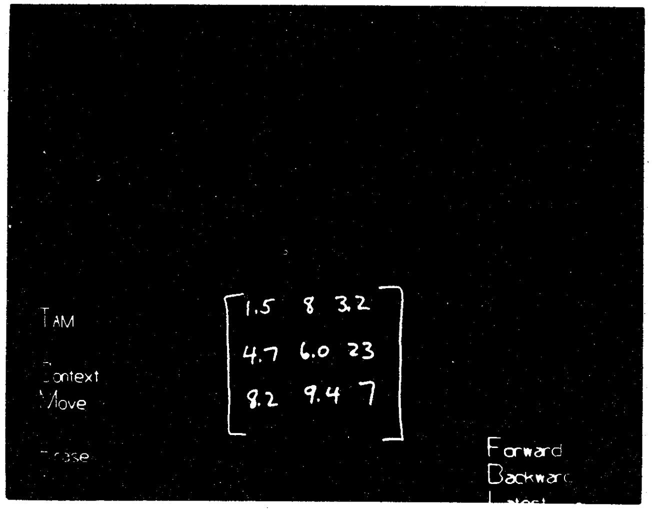
At this point, the user can instruct the computer to execute the expression by placing the tablet stylus over the area labeled 'TAM' (with the result shown in Figure 3), add to the expression, or make any necessary corrections. To make corrections, the user can call upon a number of editing operations. He may change a character by simply overwriting it with a new character. He may erase one or more characters by "scrubbing" over them as though he were scratching them out. He may also move groups of characters to open up space



$$\frac{1}{102.5+\frac{-7\sqrt{17.3}}{2\pi(6.3*10^{-4})}}$$

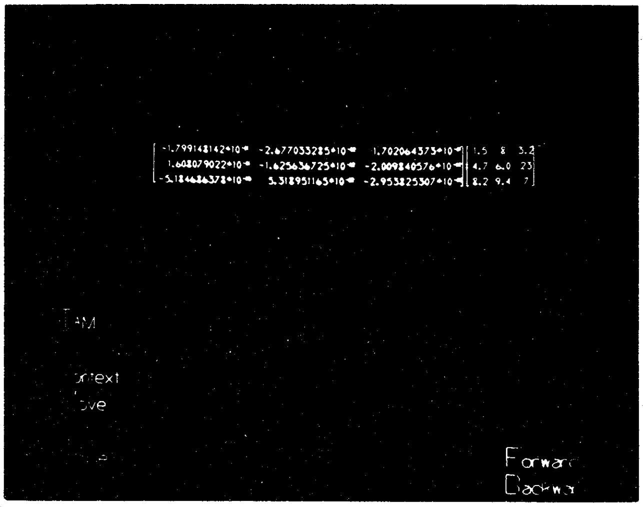
$$\frac{1}{102.5+\frac{-7\sqrt{17.3}}{2\pi(6.3\cdot 10^{-4})}}$$

Figure 2—Results of character recognition and structure analysis



$$\begin{bmatrix} 1.5 & 8 & 3.2 \\ 4.7 & 6.0 & 23 \\ 8.2 & 9.4 & 7 \end{bmatrix}$$

Figure 4—Matrix input



$$\begin{bmatrix} -1.799148142*10^{[illegible]} & -2.677033285*10^{[illegible]} & 1.702064373*10^{[illegible]} \\ 1.608079022*10^{[illegible]} & -1.625636725*10^{[illegible]} & -2.009840576*10^{[illegible]} \\ -5.184686378*10^{[illegible]} & 5.318951165*10^{[illegible]} & -2.953825307*10^{[illegible]} \end{bmatrix} \begin{bmatrix} 1.5 & 8 & 3.2 \\ 4.7 & 6.0 & 23 \\ 8.2 & 9.4 & 7 \end{bmatrix}$$

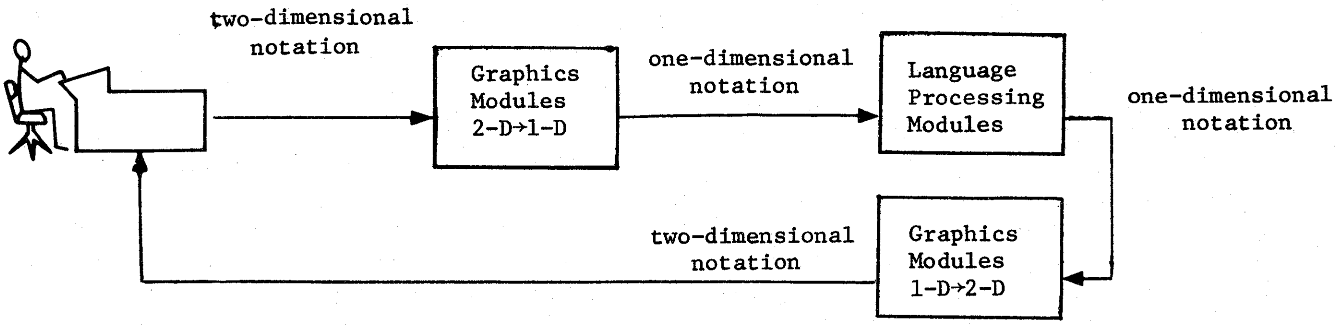
Figure 5—Matrix inversion

to insert new characters, close up an expression to delete spaces, correct errors, etc.

The system handles a wide range of mathematical notation. In all cases, the notation used for displaying the results of computation corresponds to that used for input. For instance, Figure 4 shows the input of a 3×3 matrix; Figure 5 shows the result of inverting this matrix.

SYSTEM DESCRIPTION

This section discusses the basic software modules of the TAM system. TAM handles information in two equivalent forms—two-dimensional and linear—and the system modules are distinguished into two classes, depending on the form with which they operate. *Graphics modules* operate with both two-dimensional and linear information and are used in the user-interface parts of the system. *Language-processing* modules, which handle only linear information, interpret and numerically process user requests. The overall information flow in the system is shown below:



Graphics Modules

The basic graphics modules are:

a. A recognizer for hand-printed characters;
b. A mathematics-structure analyzer, which converts a two-dimensional mathematical expression into an equivalent expression in linear infix form; and
c. A mathematics-structure builder, which converts the linear expression produced by the analyzer into the equivalent two-dimensional form.

Character Recognizer

The character recognizer serves as an input device for the rest of the system. It provides the system with the processed results of a hand-printed input in a standard form—namely, the character code assigned to the input, its size, and its position. The TAM recognizer has been designed to provide a large alphabet (in excess of 120 characters) for a given user; unlike most such efforts,[6,7,8] it is general in the sense that the user prints in his own style and only rarely must change his normal habits. This is accomplished by building a unique character dictionary for each user from samples of his own printing. Dictionary building is an interactive process, and the user may add characters or resolve ambiguities at any time. In our usage, typical character sets consist of numerals, the uppercase and lowercase Roman letters, those Greek characters that are distinct from the Roman characters, and special mathematical symbols. The character recognizer is described in Reference 11.

Mathematics-structure analyzer

The TAM mathematics-structure analyzer accepts as input a two-dimensional mathematical expression and

Two-Dimensional Form	Linear Infix Form
$\dfrac{X_i^2+Y_i^2}{Z_i^2}$	$(X[\searrow i\nearrow 2]+Y[\searrow i\nearrow 2])/(Z[\searrow i\nearrow 2])$

Figure 6—Expression representation

produces a linear infix equivalent of the expression. The form of the input is that supplied by the character recognizer: a list of character codes with associated size and position information. Only the dimensional information is converted; no conversion to Polish or tree-structure form is performed. Figure 6 shows an example of a two-dimensional form and the equivalent linear form produced by the analyzer.

Our reasons for not converting fully to a tree structure notation, as Anderson[9] does, lie in the nature of operations on an interactive time-sharing system. In general, the user constructs his expression in stages, correcting or adding to it at successive stages. We give him the results of a "trial analysis" (in two-dimensional form) at each stage, so that he can correct analyzer errors as early as possible. By not converting to a tree structure at each stage of analysis, we improve the speed and efficiency of the analyzer. However, we do sacrifice some flexibility and analysis power in comparison to Anderson's analyzer.

The analyzer accepts a wide range of mathematical notation, including subscripts and superscripts, displayed fractions, overscores and underscores on single characters and groups of characters, the $\sum$ and $\prod$ notations for summation and multiplication, the integral sign with limits, matrix and vector notation, and combinatorial notation. Many other notations, such as those used for ordinary and partial differential equations, are combinations of these basic notational devices and can also be analyzed. However, owing to the left-to-right scan of the analyzer, some notations cannot be accepted. Examples are:

${}^{4}_{3}X^{2}_{1}$	Because some subscripts and superscripts precede the main character.
$\lim\limits_{n\to\infty}$	Because the lim is handled as three distinct symbols, and the n, →, and ∞ will be treated as subscripts of the characters to the right of which they happen to fall.

In general, however, most forms of mathematical notation in common use are acceptable and are quickly and correctly analyzed.

Briefly, the analyzer operates as follows. The analyzer begins with the list of characters supplied by the recognizer, sorted into left-to-right order according to the X coordinate of the left edge of the character. The analyzer looks for a spatial relationship between a character on the input list and a reference character, which is one of the previously analyzed characters in the input list. This relationship may be either (1) one that causes the input character to be bound to the reference character as a subscript, numerator, etc., or (2) the "mainline" relation (on a roughly horizontal line) that causes the input character to become a new reference character. Characters bound to a reference character become subreference characters that can be used recursively in the analysis of an input character. This allows nested fractions and multiple levels of subscripts and superscripts.

The kinds of relationships possible between a reference character and an input character depend upon the nature of the reference character. For instance, if the reference character is a letter, the overscore, underscore, superscript, or subscript relationships are tested. If the reference character is a horizontal bar, the numerator and denominator relationships are tested. This provides a reasonably efficient search and allows a degree of error control (for instance, digits cannot have subscripts). Reference 10 contains a full description of the analyzer.

Mathematics-structure builder

To ensure that a user need deal only with two-dimensional notation, a means is provided in the TAM system to supply output to the user in two-dimensional notation. The mathematics-structure builder supplies this function. It accepts the linear expression generated by the analyzer and produces a clear, typeset-quality* two-dimensional expression. The builder is used for two purposes. First, it generates two-dimensional displays (e.g., $5.8\cdot10^9$) of the results of computation, including matrix computation. Second, it feeds back to the user the results of the analyzer's operation. Since the structure modules distinguish between brackets and parentheses supplied by the user and those generated internally, displaying only those supplied by the user, the reconstruction of a correct analysis will have the same form as the original input; the form of an incorrect analysis is usually very different from that of the input. Thus, the user need only compare the two displayed forms to verify the analysis.

The builder has two components: a scanner for finding, in succession, the main elements of the linear ex-

* That is, the information generated by the analyzer could drive an automatic typesetting device.

pression and a set of element processors, one for each of the forms used in mathematical notation. The element processors accept as input an element and its related characters (e.g., a variable and its subscripts and superscripts or a fraction bar and its numerator and denominator) and produce the two-dimensional equivalent in terms of characters and their associated size and position. Element processors also produce the location and size of the smallest possible rectangle surrounding the expression for each element. The scanner finds the main elements, calls the element processors, and strings the output of the element processors together; it also returns the size and location of the rectangle around the expression. The scanner and element processors are called recursively to analyze subparts of the expression. Details of the builder's operation are contained in Reference 10.

Language processing and notation

The TAM graphics modules accept most of the syntactical forms of conventional mathematical notation. In order for the system as a whole to behave according to the user's expectation that notation that is accepted will be taken to mean what he means by it, the system's language-processing modules must implement the semantics indicated by syntactical conventions. This, with a few exceptions, they do. (The exceptions, such as the operations of integral and differential calculus, are signaled as errors.) Despite the exceptions, the set of operations that is implemented is adequate for a large number of arithmetic-computation tasks. Some of the important semantic features of the language processors are the following:

> *Implicit multiplication*—multiplication indicated by juxtaposition of two or more variables or constants. This is a standard feature of mathematical notation; because, in TAM, all identifiers are single letters, implicit multiplication can be provided.
>
> *Implied data types and dimension*—frees the user from having to declare, in a separate statement, the data type and/or dimension of a variable. In TAM, this information is acquired from the first use of the variable, when possible. (A dimension statement for arrays does exist and is used primarily in an optional form that allows an array to be preset or cleared.)
>
> *Universal operators*—refers to the applicability of every operator to all data types for which it is meaningful. That is, any operation (such as multiplication) can apply to integers, real numbers, vectors, matrices, and any combination thereof, with no notational change or distinction.
>
> *Automatic prompting*—the ability of TAM to ask for the value of any variable that is undefined at the point of first use.
>
> *Built-in functions and constants*—some functions like sine, cosine, are called explicitly by name. Others are called by the use of the ordinary mathematical notation; e.g., A^{-1}, where A is a matrix calls the matrix-inversion function. π and e are among the built-in constants.

With a few exceptions, the notation used in TAM is standard. One exception is absolute value, for which the symbols ⟦ ⟧ were invented because, in United States usage, there is no real distinction between the printing of the digit 'one' and the printing of a vertical bar. Another exception is a loop control statement which, except for specific functions such as $\sum$ and $\prod$, does not exist in ordinary mathematics.

Entities

Quantities. Quantities in TAM are integers or mixed numbers. Quantities may be contained in variables or arrays or expressed as constants. Most storage declaration is implied by usage. Arrays are dimensioned either implicitly or explicitly.

Identifiers. Variables and array identifiers are single letters. (Note that this permits implicit multiplication.) The legal alphabet of TAM consists of Greek and Roman uppercase and lowercase letters. An identifier may be qualified (made unique) through the use of overscoring or underscoring. Legal overscore and underscore characters are:

$$- \quad \sim \quad \wedge \quad \cdot \quad \rightarrow$$

The large character set accepted by the system gives a suitably diverse set of possible identifiers. It is usually possible, for example, to copy an equation directly out of a paper or textbook and enter it into TAM.

Operators

Quantities may be manipulated through the use of operators as shown below:

$a+b$	addition
$a-b$	subtraction
$ab, a\cdot b, a*b$	multiplication
$\frac{a}{b}$, a/b	division

X^Y	exponentiation
$^n\sqrt{\ }$	n^{th} root $\sqrt{\ } => \sqrt[2]{\ }$
!	factorial
$[\![\ ]\!]$	absolute value
$\lceil\ \rceil$	ceiling
$\lfloor\ \rfloor$	floor
$\prod_{i=m}^{n}$	product
$\sum_{i=m}^{n}$	summation
$+b, -b$	unary sign
T	transpose (two-dimensional arrays only)

Each operator is usable when meaningful. With few exceptions (for example, transpose applies only to two-dimensional matrices; $(-3)!$ is signaled as an error), all operators are usable to manipulate quantities, variables, and arrays. The operators are legal when applied to arrays when an acceptable matrix or vector operation is defined. One-dimensional arrays are stored and treated as row vectors ,with one exception: in multiplication, if one or both operands are vectors, the operand on the left is treated as a row vector and the operand on the right is treated as a column vector. The multiplication performed is the dot product.

Statements

There are three distinct TAM arithmetic-computation statements: assignment, function definition, and loop. There are also some built-in functions.

Assignment. The assignment statement is used to set identifiable variables or arrays, presumably for use in subsequent statements. An assignment statement is of the form:

identifier ← expression

The expression may consist of any legal manipulation of quantities.

Function Definition. The TAM user may define frequently used arithmetic expressions as functions; he may then call upon these functions when necessary. Functions, of course, return values. The function definition and call may contain parameters. Both the function expression and the actual parameters of the call may contain calls to other functions. The function-definition statement has the form:

$f_n(p_1,p_2,\ldots p_m) =$ expression

where f is a legal identifier, n is an optional alphabetic or numeric qualifier, and the p_i are optional parameters. The expression may involve any legal manipulation of quantities. The identifier f, once it has been used as a function name, defines a class of functions f_n and cannot be used later as a variable or array identifier. The functions in class f are distinguished from one another through the use of the qualifier n. (For example, $\bar{G}_1(X)=X^2$ and $\bar{G}_2(X)=\sqrt{X}$ are two functions in class $\bar{G}$; $\bar{G}\leftarrow 3$ is an illegal statement; $\hat{G}$ and G are not in class $\bar{G}$.) As many function classes as desired may be defined. The optional parameters, p_i, must be legal identifiers and may be used as parameters in many function definitions and also as variables or array names or as function classes.

Loop Control. A statement may be iterated by following it with loop-control information. Loops may be nested to any level, but each loop variable in the nest must be unique. Loop control may be specified in three forms:

Loop Control, Form 1

Statement: $i=m, \ldots, n$
where i is the loop variable (an identifier of a simple variable whose value will be incremented by one for each iteration of the statement), m is the initial value for i, and n is the terminal value for i. m and n may be any legal expressions that yield single numeric values. The iteration is complete when i exceeds n. The statement iterated may, but need not, contain references to i.

Loop Control, Form 2

Statement: $i=m_1, m_2, \ldots, n$
where i is the loop variable, m_1 and m_2 are the first two values for i as the statement is iterated, m_2-m_1 defines the loop increment (or decrement), and n is the terminal value. m_1, m_2 and n may be any legal expressions that yield single numeric values. The iteration is complete when i exceeds (or becomes less than) n. The statement iterated may, but need not, contain references to i.

Loop Control, Form 3

Statement: $i=m_1, m_2, m_3, m_4, \ldots m_n$
where i is the loop variable and the m_j are successive settings for i each time the statement is iterated. (The elipsis (...) shown is not a part of the loop-control form, as it is in the two previous forms, but is included to indicate that the list

m_j is of user-determined length.) The loop terminates after the statement has been executed for $i = m_n$. The statement may, but need not, contain references to i.

Built-In Functions. TAM includes a set of built-in functions that the user can activate by including one of the names given below (along with an appropriate parameter) within any context in which a function call is permissible. The available functions are:

Name and Parameter	*Definition*
sin (x)	sine (x)
cos (x)	cosine (x)
tan (x)	tangent (x)
cot (x)	cotangent (x)
arctan (x)	arctangent (x)
$\tan^{-1}(x)$	arctangent (x)
ln (x)	natural logarithm (x)

In expressing the name, any combination of uppercase and lowercase Roman letters is permissible; e.g., Ln≡ln≡LN.

The built-in functions also include those called by notational devices: square root, exponential, matrix inverse, matrix transpose, and factorial.

CONCLUSION

TAM is an experimental system designed and built to test the usefulness of, and problems associated with providing, natural man-machine communication in the context of problem solving by the non-programmer physical scientist or engineer. We feel that TAM has demonstrated the usefulness of man-machine communication. It provides considerable computation power in a simple, flexible way; learning how to use it requires very little time, even if one has not programmed a computer; and remembering how to use it is easy.

More importantly, however, designing TAM has helped us to identify the real problems of working with natural notation. Fundamentally, any natural notation, including mathematics, is ambiguous and context dependent. As one result of this, TAM does not contain built-in complex arithmetic capabilities, principally because it is difficult to resolve the use of the letter i as an integer (index of summation), mixed number (arbitrary variable), or $\sqrt{-1}$, without reference to the global context in which it is being used. Therefore, the use of natural notation requires some solution of the ambiguity problem.

Two solutions are possible. One, adopted in most programming languages, is to require explicit specification by the user of the things he wants to use so that there can be no ambiguity; this specification must be provided for each new program. The other, which we are beginning to explore, is to assume that the user works for a period of time within some specific computational context, with its own defined notations, functions, and data, and that it should be possible to establish in the computer, more or less implicitly, a contextual framework within which the user works as long as he continues with a specific problem or set of problems. The user might start with a general context appropriate to his general problem area. To this he could add his notational devices and functions, much as he might do in defining notation and functions when writing a paper. A contextual framework could also provide a data-handling capability. The notions of information storage and retrieval are foreign to mathematics, and the volume of numeric data required for many useful and interesting problems is beyond the reasonable capacity for entry from a data tablet. The context system could provide a computational capability over a defined data base, thus providing a simple way to process and reprocess data.

We have, basically, just begun to explore the possibilities opened up by the freedom of two-dimensional input. We foresee that the TAM system, and others like it, are forerunners of a new capability and flexibility in natural man-computer interaction. We look forward to the day when a user, in any field, can sit down and communicate with a computer in the language of his choice or invention.

ACKNOWLEDGMENT

The continuing progress made on this project owes a great deal to many individuals for support, constructive criticism and direction. In particular, Mort Bernstein for his overall guidance; Jean Igawa, for her work on the character recognition; John McGahey, for his work on the graphics modules; and Joan Bebb and Jean Saylor, for the design and implementation of the language processor.

REFERENCES

1 W R DE HAAN
Automatic graphic schematic drawing program
Proceedings Third SHARE Design Automation Workshop May 1966

2 W J BOUKNIGHT K KELLEY
An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources
Proceedings AFIPS 1970 Spring Joint Computer Conference Vol 36 pp 1-10

3 R M BAECKER
Picture driven animation
Proceedings AFIPS 1969 Spring Joint Computer Conference Vol 34 pp 273-288

4 R R LINDE et al
The ADEPT-50 time-sharing system
Proceedings AFIPS 1969 Fall Joint Conference Vol 35 pp 39-50

5 L A GALLENSON
A graphic tablet display console for use under time-sharing
Proceedings AFIPS 1967 Fall Joint Computer Conference Vol 31 pp 689-695

6 T L DIAMOND
Devices for reading hand-written characters
Proceedings Eastern Joint Computer Conference December 1957 pp 232-237

7 G F GRONER
Real-time recognition of hand-printed text
Proceedings AFIPS 1966 Fall Joint Computer Conference Vol 29 pp 591-602

8 W TEITLEMAN
Real-time recognition of hand-drawn characters
Proceedings AFIPS 1964 Fall Joint Computer Conference Vol 26 pp 559-576

9 R H ANDERSON
Syntax-directed recognition of hand-printed two-dimensional mathematics
Presented at ACM 1967 Symposium on Interactive Systems for Experimental Applied Mathematics

10 M I BERSTEIN
On-line, interactive parsing and programming; final report for Phase III
System Development Corporation Document TM-4582 August 1970

11 M I BERNSTEIN T G WILLIAMS
A two-dimensional programming system
System Development Corporation Santa Monica California
Information Processing 68-North-Holland Publishing
Company-Amsterdam (1969)

Debugging PL/I programs in the multics environment

by B. L. WOLMAN

Honeywell Information Systems
Cambridge, Massachusetts

INTRODUCTION

One of the popular misconceptions concerning PL/I is that programs written in PL/I are necessarily inefficient and hard to debug. Several years experience with the Multics PL/I compiler running on the Honeywell 645 has shown that in spite of the apparent complexity of the PL/I language, PL/I programs are easily debugged in the Multics environment, even by novice users who are newcomers to PL/I and are unfamiliar with the Honeywell 645. In most cases the user can debug his program symbolically without having to refer to a listing of the generated instructions or add debugging output statements to the program. This is due to a number of factors:

- the run-time environment provided by the system.
- the implementation of PL/I.
- the availability of a variety of powerful debugging facilities.

THE ENVIRONMENT

The use of PL/I as the principal tool for programming by users of Multics was envisioned at the very start of the project. Features which are required by PL/I such as a stack, pointer variables, conditions, and a recursive call/return mechanism are all provided and are directly supported by the system hardware and/or software. Consequently, the basic Multics environment is ideally suited to the needs of PL/I programs. In fact, nearly all of Multics itself is coded in PL/I and executes in this self-maintained environment.[1–5]

The Multics system currently provides the user with a virtual address space of over 1000 segments of 65536 words each (some changes now in progress will increase the maximum size of a segment to 262144 words). Access to these segments is by means of PL/I pointer variables which contain a segment number, a word offset, and a bit offset. There is a direct correspondence between PL/I pointers and virtual addresses in Multics; PL/I pointer values may be loaded into the addressing registers of the 645 by a single machine instruction. An attempt to use a pointer whose value is the PL/I null pointer causes a condition to be signalled.

The PL/I stack is maintained for each user as a series of contiguous frames (block activation records) within a single segment. A register is dedicated by the system to point at the stack frame of the procedure being executed. Multics defines a system-wide standard call/return sequence which is relatively efficient. Stack frames can be obtained and released by executing a few instructions.

Procedure segments in Multics are normally pure and sharable. Access to procedure and data segments is set by Multics access control commands and checked by the hardware at each instruction and data reference. If a user does not have appropriate access to a segment, or if any other error such as an attempt to divide by zero happens, a machine fault occurs. This fault is turned into a PL/I condition (e.g., "accessviolation" or "zerodivide") and is signalled by the PL/I condition mechanism. All but a few catastrophic errors are handled in this manner.

Multics provides a default error on-unit which is invoked if the user has not established an on-unit for a specific condition. In most cases, the default on-unit prints an appropriate error message (which may include information as to probable causes for the error) and calls the command processor to read a command from the user's input stream. The stack chain of calls leading up to the fault is preserved; in many cases the user's program can be restarted.

In Multics there is no real difference between a command and a program written by the user: both are PL/I procedures. Any program written in PL/I following command argument conventions may be invoked as a "command".

When the user types a command line of the form

edit alpha beta

the Multics command processor searches a specified set of directories for a procedure named "edit" and issues the equivalent of the PL/I statement

call edit("alpha", "beta");

The procedures found in the system directories are the "commands" and utility procedures normally available to Multics users. Since the user can change the search rules used by the system, he can tailor his own command set if he chooses.

THE IMPLEMENTATION

The implementation of PL/I in Multics is particularly complete and has few restrictions.[6,7] The only omission of any consequence is tasking. The Multics implementation allows:

- arbitrary pointer qualification including chains of locators and use of functions as qualifiers.
- adjustable data with no restrictions. Arrays may have any number of adjustable bounds. Structures may have any number of adjustable members.
- operations on aggregates.
- functions which return values whose length or bounds are not known at the time the call is made, e.g., returns(char(*)) or returns((*) fixed bin).
- entry variables.
- recursive procedures at no extra cost.
- full stream and record I/O.
- all data types including complex and decimal.

Since the implementation is so complete, the programmer does not have to worry about what features are or are not available to him. The ability to use the full language reduces the amount of code the user has to debug by increasing the amount of work handled by the run-time support system provided by the compiler.

The Multics PL/I compiler produces efficient object code, even when measured against the best efforts of experienced hand coders using assembly language. The availability of a compiler which generates efficient programs greatly reduces the user's desire to want to switch to assembly language for reasons of efficiency. This is particularly important in Multics because of the richness of the machine instruction set (512 instructions and 64 types of address modification) and the complexity of the system environment from the view point of an assembly language coder.

Multics PL/I makes use of a separate "operator segment" which contains assembly language coding for about 50 commonly used functions such as string moving, complex multiplication, and the index operator, as well as tables of constants for masking, shifting, storing characters, etc. This segment is shared by all PL/I programs. Communication with the operator segment is by means of a work area in a standard position in each stack frame. The operator segment is entered by a short sequence of instructions which loads certain machine registers with parameters and then jumps directly into the operator segment at a known location. The use of the operator segment reduces the cost of PL/I programs by reducing their size and by reducing paging activity.

If a begin block or internal procedure block does not declare any automatic variables with adjustable bounds or sizes and can only be entered by first entering its parent block, then the block is said to be "quick". The Multics PL/I compiler does not use a separate stack frame for such blocks. Instead, they share the stack frame of their parent block. The overhead of calling a quick block, exclusive of the cost of preparing the argument list, is only three instructions: one each at call, entry, and return. The cost of a quick procedure is also reduced because automatic storage in the parent block can be addressed directly.

The availability of a really inexpensive mechanism for internal procedures means that users can write them without having to concern themselves with efficiency. The artifice of using label variables and goto statements so that a block of code can be executed efficiently from a number of places is not necessary.

The compiler makes no restrictions on the format of structures. This is important, since programmers can choose a structure description that is appropriate for the problem they are trying to solve without having to consider its acceptability to the compiler. However, it *is* possible for a user to specify a structure which causes the compiler to generate very expensive accessing code. There are a few "common sense" rules users can follow if they are concerned about the efficiency of their programs.

Extensive error checking is done during compilation; there are nearly 500 possible error messages. Except for a few cases of multiple, related errors within a single statement the Multics PL/I compiler normally finds most errors in a single run. It is infrequent that a user will correct a set of source errors and recompile his program only to receive another batch of error messages. Errors are reported on the user's console as they are discovered; the printed message normally includes the source for the offending statement.

The listing generated by the compiler is designed to be printed by a high-speed line printer but is formatted so that items of interest to the user can be easily located in the listing segment by inspecting it with an on-line editor. The user can control the amount and level of detail of information placed in the listing.

DEBUGGING FACILITIES

Multics provides a number of special commands which aid user debugging. There is a powerful breakpoint debug command, a facility for tracing procedure calls, and tools which help the user determine the operating characteristics of his programs. There are several options that the user can specify when he uses the PL/I compiler to cause it to generate additional information for use by debugging commands. Of these, only the "profile" option causes any change in the code generated by the compiler.

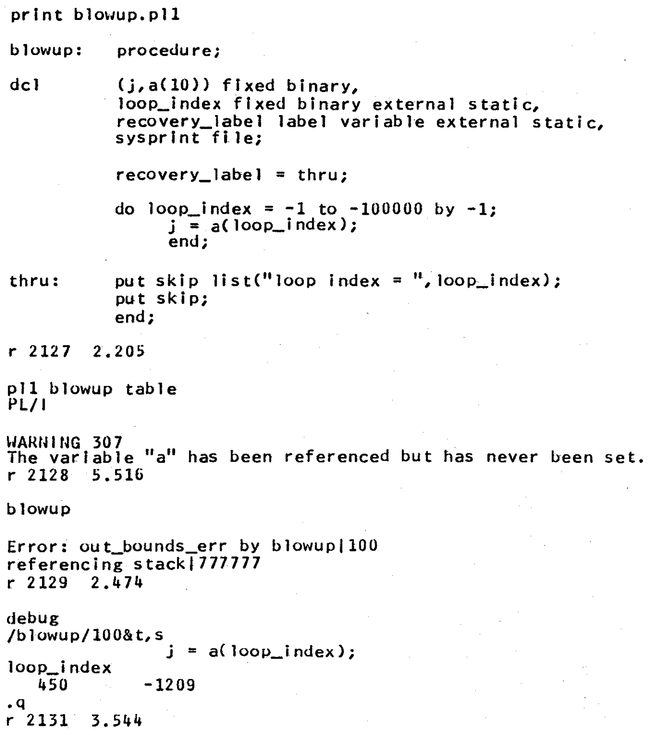
The run-time symbol table

The PL/I compiler and the system debug command cooperate to allow the user to debug his program symbolically. The compiler normally generates a run-time symbol table only if "get data" or "put data" statements are used in the source program. The compiler can be instructed, however, to generate a "full" symbol table which includes all identifiers in the source program.

Each entry in the run-time symbol table describes an identifier in the user's program giving its name, storage class, location, size, bounds and other information needed to access the identifier. Information is available about the block in which the identifier is defined as well as its relationship to other members of the strucutre to which it belongs.

The run-time symbol table facility is much more powerful than it needs to be just to support data directed I/O.

- Parameters, defined, and based variables can all be represented in the table. When a variable is declared based on a specific pointer, e.g., "dcl a based(p)", information is kept which allows the address of that pointer to be obtained at run-time.
- The size, offset, bounds, multipliers, or virtual origin of any identifier can be any arbitrary expression. This is necessary for the representation of based variables.
- References to identifiers in the user's program from data directed input or from requests to the system debugger need not be fully qualified. The same algorithm used by the compiler to resolve partially qualified names is also used by the support program which searches the run-time symbol table.



```
print blowup.pl1

blowup:   procedure;

dcl       (j,a(10)) fixed binary,
          loop_index fixed binary external static,
          recovery_label label variable external static,
          sysprint file;

          recovery_label = thru;

          do loop_index = -1 to -100000 by -1;
               j = a(loop_index);
               end;

thru:     put skip list("loop index = ",loop_index);
          put skip;
          end;

r 2127  2.205

pl1 blowup table
PL/I

WARNING 307
The variable "a" has been referenced but has never been set.
r 2128  5.516

blowup

Error: out_bounds_err by blowup|100
referencing stack|777777
r 2129  2.474

debug
/blowup/100&t,s
                 j = a(loop_index);
loop_index
   450        -1209
.q
r 2131  3.544
```

Figure 1—The PL/I condition mechanism is used for most errors, including those defined by Multics. In this example, the program generates a fault by looping until it runs off the front of the stack. The default error on-unit prints the location at which the fault occurred (100 in blowup) and the location being referenced (−1 in the stack). The program was compiled with a run-time symbol table, so the Multics debug command may be used to print the source for the line in which the fault happened. The request syntax accepted by debug is designed to minimize typing: the request specifies segment blowup, location 100 in the text section, and source line output. The value of a variable may be obtained merely by typing its name; the response gives the address of the variable (450 in the static data segment) as well as its value (−1209).

The run-time symbol table is generated at the end of the object segment and is shared by all users of the segment. If it is not used during execution, there is no overhead required to support it: the pages it occupies will not be brought into core memory; no code is required to initialize it. After the program has been debugged, the run-time symbol table can be eliminated from the object segment without having to re-compile it.

The compiler will also generate a "map" of the object program when a full symbol table is requested by the user. This map is a table, placed at the end of the object segment, giving information about the location in the object segment of each source statement. The availability of this table means that the user can refer to his object program by source line number, e.g., to set a breakpoint at a specific line number. Similarly, the system debugger can tell him the line number corresponding to a given location in the object program. In fact, as is demonstrated in Figure 1, the debug command can print the source line that corresponds to the object location.

The debug command

The command "debug" can be invoked at any time; for example, after an error condition has been signalled for which no on-unit exists. It may also be called directly from the user's program. It accepts requests from the user for actions such as examining some location in the virtual memory or printing a trace of the chain of calls in the user's stack. It is aware of the different PL/I data types, so variables in the object program may be displayed in the format appropriate to their type.

When a program has been compiled with a run-time symbol table, the user can refer to it symbolically, either with identifiers defined in the program or by the line number on which a statement begins. For example, if the user's program was dealing with a two-dimensional based array of integers, he could change one of the elements in the array by entering the request

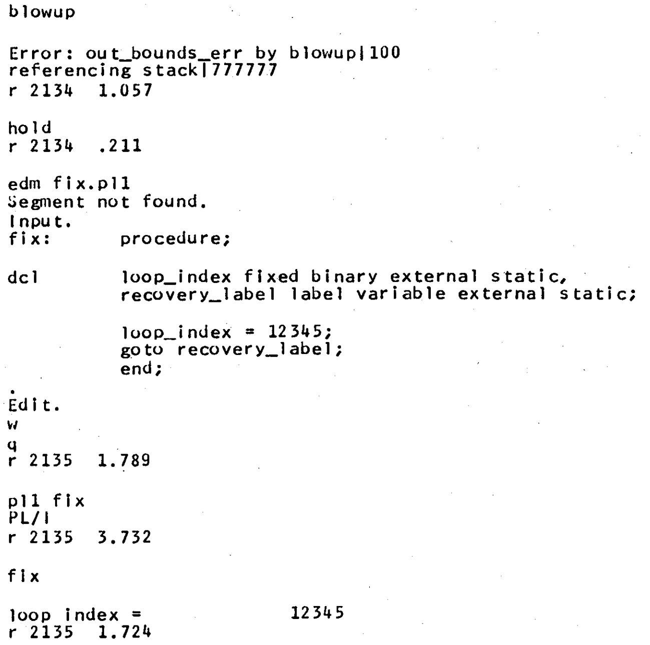
$$p\rightarrow x(i+5, j-2)=3$$

which takes the form of a PL/I style assignment. The addresses of "p", "x", "i", and "j" would be obtained from the symbol table. Any of the identifiers in this example could be part of a structure.

The debug command can also be used with PL/I programs when a run-time symbol table is not available. In this case, the user must refer to the compilation listing of his program in order to determine the location at which a variable is stored or at which a given statement starts.

The debug command has other features which let the more experienced user examine or alter the values in a machine register or display the status of the machine at the time a fault occurred. These facilities are not normally needed if a symbol table is available.

The debug command also lets the user set conditional or unconditional breakpoints in object segments. When the breakpoint instruction is executed, the debug program gains control. If the condition associated with the breakpoint is satisfied, a message is printed; at this point the user can enter requests to debug. One of the actions available is to continue execution from the point of the break. The user may associate with each break a set of debug requests which are to be automatically executed whenever the break is encountered; thus, for example, the user might use the break mechanism to "insert" a (very simple) PL/I assignment statement into his program. There is a mode of execution available with debug which lets the user run his program one PL/I statement at a time.



```
blowup

Error: out_bounds_err by blowup|100
referencing stack|777777
r 2134  1.057

hold
r 2134  .211

edm fix.pl1
Segment not found.
Input.
fix:      procedure;

dcl       loop_index fixed binary external static,
          recovery_label label variable external static;

          loop_index = 12345;
          goto recovery_label;
          end;

.
Edit.
w
q
r 2135  1.789

pl1 fix
PL/I
r 2135  3.732

fix

loop index =                12345
r 2135  1.724
```

Figure 2—When a fault occurs, the complete status of the executing program may be preserved. The "hold" command causes Multics to retain the chain of stack frames (block activation records) up to the current frame until the user issues an explicit "release" command. In this example, the user inputs and compiles a small procedure to fix up the loop index that caused the bounds violation in the example of Figure 1. The program blowup is reactivated by a non-local transfer of control to the external label variable and completes normally. The same change of the loop index and re-start of blowup could also be done using only the debug command.

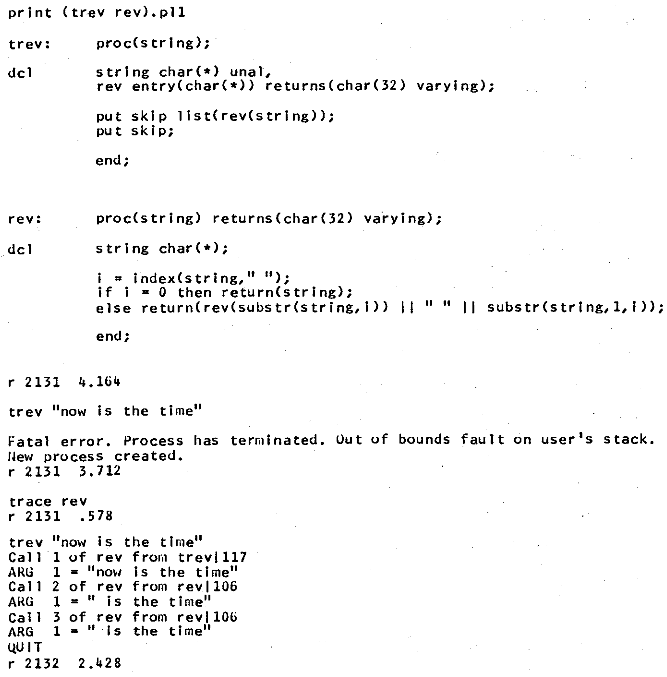
An object program may have more than one break set in it; similarly, more than one program may have active breakpoints. Facilities are available in debug for listing and altering breaks. Setting a break involves changing the object program, so breakpoints remain active until explicitly removed by the user. Breakpoints should not be used when other users are sharing the segment.

There is an "escape" facility which causes debug to pass the line typed by the user to the Multics command processor instead of treating it as a request. This is a *very* powerful feature since it allows the user to invoke *any* series of Multics commands (or any of his own programs) without having to leave the debug command. He could, for example, run a special program to display the values of the static variables used by the program he is trying to debug. If he did not have such a program, he could input it, compile it, and test it while preserving the complete status of the program he was originally debugging.

The ability to "escape" back to the full Multics system to execute any series of commands is generally available in any command such as the editor that interacts with the user. As is shown in Figure 2, the "hold" command may be used to preserve the execution environment after a fault.

The trace command

The command "trace" lets the user monitor all calls to a specified set of external procedures. Trace modifies the standard Multics procedure linkage mechanism so that whenever control enters or leaves one of the procedures specified by the user, a debugging procedure is invoked. The arguments given to the debugging procedure by trace enable it to obtain the values of the arguments and return point of the procedure being called. The user can also provide his own debugging procedures instead of the one supplied as a default by the tracing package.



```
print (trev rev).pl1

trev:     proc(string);

dcl       string char(*) unal,
          rev entry(char(*)) returns(char(32) varying);

          put skip list(rev(string));
          put skip;

          end;


rev:      proc(string) returns(char(32) varying);

dcl       string char(*);

          i = index(string," ");
          if i = 0 then return(string);
          else return(rev(substr(string,i)) || " " || substr(string,1,i));

          end;

r 2131  4.164

trev "now is the time"

Fatal error. Process has terminated. Out of bounds fault on user's stack.
New process created.
r 2131  3.712

trace rev
r 2131  .578

trev "now is the time"
Call 1 of rev from trev|117
ARG  1 = "now is the time"
Call 2 of rev from rev|106
ARG  1 = " is the time"
Call 3 of rev from rev|106
ARG  1 = " is the time"
QUIT
r 2132  2.428
```

Figure 3—The flow of control in to and out of any external procedure may be monitored with the Multics debugging procedure trace. In this example, trev is a driver program which calls procedure rev to reverse the words in a string specified by the user when trev is called. rev is coded as a recursive procedure; it contains a bug which causes infinite recursion. The "fatal error" occurs when there is no room left in the stack segment for a new frame. The reason for the infinite recursion becomes obvious when trace is used.

The action taken by the default trace debugging procedure is to print a message on the user's console whenever control enters or leaves one of the procedures being traced. There are a number of options which the user can specify to request such actions as printing the arguments (at entry, exit, or both) or stopping (at entry, exit, or both). The user can control the frequency with which the tracing message is printed, e.g., every 100 calls after the 1000th call. He can also specify the maximum recursion depth he wishes to see. The user can also request that the tracing message be printed only if the contents of some specified location in the virtual memory has changed. The default trace debugging procedure "stops" the execution of the user's program by calling the debug command; this makes all of the facilities of debug available to the user. An example of the use of trace is presented in Figure 3.

The user may start tracing a procedure at any time, even if it has already been executed. Tracing may be removed at any time; subsequent calls of the procedure will execute normally. Any procedure which uses the standard Multics calling sequence may be traced without interfering with other users who may be sharing the segment.

Determining program efficiency

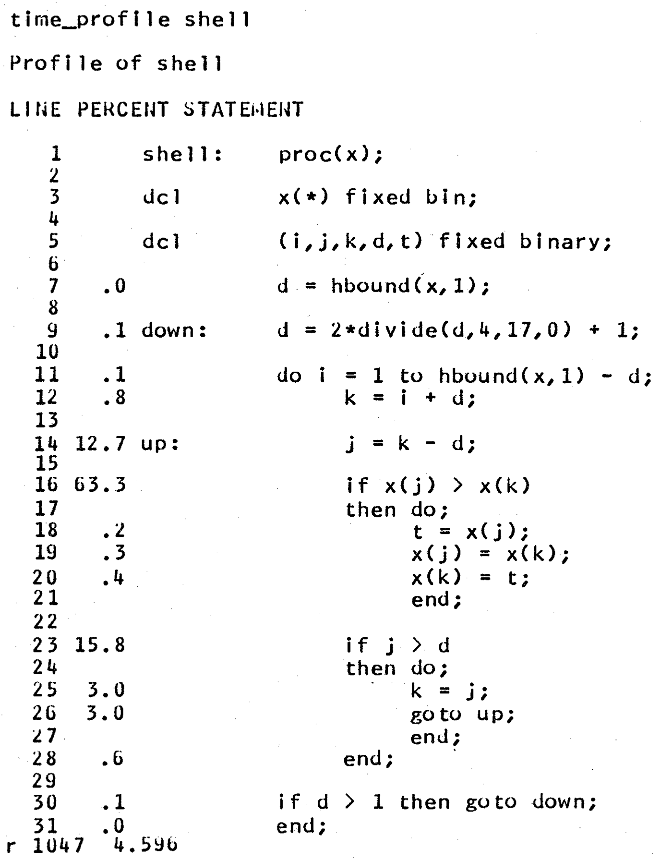
The two debugging packages debug and trace which we have just discussed help the user find errors which prevent his program from running properly. There is another class of errors which are much harder to find. These are usually flaws in the program design (or perhaps in its implementation) which cause the program to run correctly but to take much longer to execute than it should. Simply locating the largest statement in the program or the biggest procedure is not sufficient to locate the causes of program inefficiency because that statement or procedure may be executed only once; the real offender may be some small statement which gets executed very frequently. Without detailed knowledge of program flow during execution, instruction counts alone are not much good.

The cost of executing a specified procedure, either for a single call or a total of many calls, can be determined by using the "meter" option of the trace command. This causes trace to read the system clock when control enters or leaves the specified set of procedures. The clock counts in microsecond steps, so high resolution is possible.

Once a procedure has been found to be inefficient, its operating characteristics can be examined by recompiling it with the PL/I "profile" option.[8] This option causes the compiler to generate in the internal static data area a table which contains an entry for each statement in the source program; the table entry contains information about the source line as well as a counter which starts out as zero. Each statement in the program is modified to start with an instruction to add one to the counter associated with the statement.

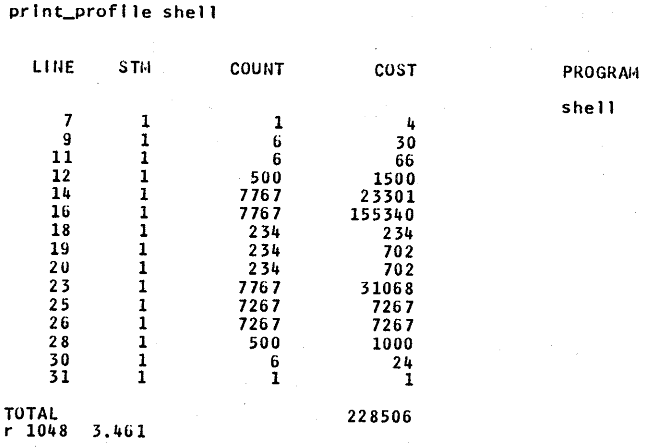
After running a program compiled with the "profile" option, the user can determine the number of times each statement in the program was executed. The table entry contains the raw cost of the statement measured in instructions, so the user can determine both the absolute total cost for the statement as well as its relative cost compared to other statements.

A number of different tools have been developed for presenting the information available in the profile table. Figure 4 shows the source for a small procedure printed by a program which computes the percentage of the total time spent in each statement. Figure 5 shows the same profile information presented in another format.



```
time_profile shell

Profile of shell

LINE PERCENT STATEMENT

   1       shell:    proc(x);
   2
   3       dcl       x(*) fixed bin;
   4
   5       dcl       (i,j,k,d,t) fixed binary;
   6
   7    .0           d = hbound(x,1);
   8
   9    .1 down:     d = 2*divide(d,4,17,0) + 1;
  10
  11    .1           do i = 1 to hbound(x,1) - d;
  12    .8                k = i + d;
  13
  14  12.7 up:            j = k - d;
  15
  16  63.3                if x(j) > x(k)
  17                      then do;
  18    .2                     t = x(j);
  19    .3                     x(j) = x(k);
  20    .4                     x(k) = t;
  21                           end;
  22
  23  15.8                if j > d
  24                      then do;
  25   3.0                     k = j;
  26   3.0                     goto up;
  27                           end;
  28    .6                end;
  29
  30    .1           if d > 1 then goto down;
  31    .0           end;
r 1047  4.596
```

Figure 4—The execution profile of a Shell sort routine after having sorted the descending sequence 999, 998, . . . , 0 into ascending order. Each statement is labelled with the percentage of the total execution time spent in that statement. The profile tells us that the algorithm is quite good since unnecessary interchanges were not often done.



```
print_profile shell

  LINE     STM        COUNT        COST                PROGRAM

                                                       shell
     7       1            1           4
     9       1            6          30
    11       1            6          66
    12       1          500        1500
    14       1         7767       23301
    16       1         7767      155340
    18       1          234         234
    19       1          234         702
    20       1          234         702
    23       1         7767       31068
    25       1         7267        7267
    26       1         7267        7267
    28       1          500        1000
    30       1            6          24
    31       1            1           1

TOTAL                            228506
r 1048   3.461
```

Figure 5—Another presentation of the execution profile of the procedure shown in Figure 4. The cost is measured in number of instructions executed.

The paging characteristics of a program can be measured by using the "page trace" facility. The Multics paging mechanism maintains a buffer for each user in which the system records the segment number, page number, and time of occurrence for each of the last few hundred page faults taken by the user's process. A command is available which formats the information kept by the system.

DIFFICULTIES

As might be expected, there are problems associated with debugging PL/I programs in Multics. Most of these problems are minor and have the effect of requiring the user to know more about the internal workings of Multics than he might otherwise have to know.

The most difficult problem occurs when a program in the user's process commits an error so severe that the system cannot continue running the process. An example of such an error is using up the entire stack segment (perhaps because of unlimited recursion). When the system detects an error of this magnitude, it prints a message such as:

> Fatal Process Error. Out of bounds fault on user's stack.

and creates a new process, thereby erasing all information about the old process.

This type of error can be very difficult to find, because no information is available to the user about where it occurred. Future versions of Multics will alleviate this problem by allowing the user to retain information about the old process. The system will also be changed to detect when the user is near the end of his stack; when this occurs, a special "stack" condition will be signalled.

COMPARISON WITH OTHER WORK

PL/C[9] and the IBM Checkout Compiler[10] are approaches to the problem of debugging PL/I programs in which a special compiler is used during the debugging phase. Extra checking is done at run-time to catch programming errors such as the use of undefined variables. No particular effort is made to generate good object code since it is assumed that the program will be re-compiled with a production compiler after having been debugged with the special compiler.

An advantage of this approach is that a great deal of information about the original source program may be preserved at run-time, thus allowing good diagnostics. A debugging compiler can often check for errors whose detection would be intolerably expensive for a production compiler, e.g. a mismatch between a based variable and the object identified by the pointer value. The Checkout Compiler allows the user to make incremental symbolic additions to his program, a very desirable feature.

A disadvantage of using a special complier is that two compilers are involved in the debugging process and therefore two sets of compiler bugs. Another disadvantage is that meaningful figures on program performance are hard to obtain.

Multics provides a single PL/I compiler which is used by all programmers, whether novice or expert. Extra checking (other than that defined as part of the PL/I language) is not done at run-time. The run-time symbol table and the map of the object program let the user refer to his program symbolically. Since a production compiler is being used, accurate figures on program performance are available.

A "program" in Multics often consists of a number of separately compiled procedures; the Multics PL/I compiler, for example, consists of 181 procedures comprising over 137,000 instructions. Because of the poor runtime performance normally available with a special debugging compiler, it is doubtful whether such a large collection of procedures could be successfully implemented using a debugging compiler. Since a special compilation is not required for their use, the Multics debugging tools debug and trace may be successfully used in finding bugs in production software. Even if a module could be re-compiled with a debugging compiler, the resulting object program would not be the same as the one which failed.

EXDAMS[11] is a powerful debugging tool which uses a pre-processor to modify the original source program before compilation. Calls to special monitoring procedures are inserted at points of interest in the program. During execution a record is kept of the complete execution history of the program. This allows the programmer to easily determine the point at which a given variable changes, for example. This sort of debugger would be useful, even in Multics, when a program is first being debugged; its usefulness is limited by the fact that a special compilation is required.

Evans and Darley[12] discuss source language debugging of higher-level languages. They present a number of principles which they believe are important. The Multics debugging commands satisfy most of their criteria:

1. The user has flexible control over the execution of his program. The program may be run in steps which range from a single procedure call, through a single statement, down to a single instruction.
2. The data being operated on may be examined and altered at any time and this may be done in the PL/I notation.
3. The conventions of the debugging language are to a large extent designed to minimize typing. (It is only fair to point out that the Multics debug command has been accused of being overly terse.)

The area in which Multics falls short of the features desired by Evans and Darley is the lack of the facility for incremental compilation.

ACKNOWLEDGMENTS

The Multics PL/I compiler was designed and implemented by R. A. Freiburghouse, the author, G. D. Chang, and J. D. Mills; significant contributions were also made by P. A. Belmont, P. A. Green, and A. C. Franklin. The Multics debug command was written by S. H. Webber. The trace command was written by the author. Many other members of the Honeywell and M. I. T. staffs, notably M. B. Weaver, D. Bricklin, and D. P. Reed, have made important contributions to easing the process of debugging PL/I programs in Multics.

REFERENCES

1 E I ORGANICK
The multics system: An examination of its structure
MIT Press Cambridge Massachusetts 1972

2 A BENSOUSSAN C T CLINGEN R C DALEY
The multics virtual memory: Concepts and design
Comm ACM 15 5 May 1972 pp 308-318

3 R C DALEY J B DENNIS
Virtual memory, processes and sharing in multics
Comm ACM 11 5 May 1968 pp 306-312

4 F J CORBATÓ J H SALTZER C T CLINGEN
Multics—The first seven years
AFIPS Conf Proc 40 1972 SJCC AFIPS Press 1972 pp 571-583

5 *Multics programmers' manual*
Honeywell Document AG90-93 1972

6 R A FREIBURGHOUSE
The multics PL/I compiler
AFIPS Conf Proc 35 1969 FJCC AFIPS Press 1969 pp 187-199

7 R A FREIBURGHOUSE
The multics PL/I language
Honeywell Document AG94 1972

8 D E KNUTH
An empirical study of Fortran programs
Stanford University Computer Science Department Report CS-186

9 H L MORGAN R A WAGNER
PL/C:—The design of a high-performance compiler for PL/I
AFIPS Conf Proc 38 1971 SJCC AFIPS Press 1971 pp 503-510

10 *IBM System/360 operating system: PL/I checkout compiler*
IBM form number GC33-0003 1971

11 R M BALZER
EXDAMS—EXtendable Debugging and Monitoring System
AFIPS Conf Proc 34 1969 SJCC AFIPS Press 1969 pp 567-580

12 T G EVANS D L DARLEY
On-line debugging techniques: A survey
AFIPS Conf Proc 29 1966 FJCC AFIPS Press 1966 pp 37-50

Data structures in the extensible programming language AEPL*

by E. MILGROM**

New York University
New York, New York

and

J. KATZENELSON***

Technion-Israel Institute of Technology
Haifa, Israel

INTRODUCTION

The extensible programming language AEPL has been designed as a tool for the implementation of a large class of problem-oriented languages or languages for specific applications. The reason for such a goal is that we believe that there exist numerous areas of human interest generating problems which can be solved with the aid of a computer. We think also that to be able to approach these problems using languages which are close to the terminology and the methodology of the respective areas is a significant advantage: it enables a user to think in familiar terms and it liberates him from the burden of extraneous detail. This has been the reason for the uneconomic proliferation of a large number of programming languages, each more or less well adapted to the solution of a particular class of problems (see for instance Sammet's book[18] for a survey of a number of problem-oriented languages). Extensible languages propose to cover wide areas of application at lesser cost and greater convenience. A detailed description of a large number of current extensible languages and systems can be found in a report by Solntseff,[21] together with an extensive bibliography of the area. Many extensible language schemes have been described in detail.[2–5,7,9,10,16,19]

At the present time, we do not believe that the existing extensible languages can reasonably claim to replace all existing general-purpose and special purpose languages, mainly for reasons of efficiency. We concede therefore that the usefulness of AEPL will be greatest for application areas which do not warrant the cost of a specially written compiler and where the matter of efficiency is relatively unimportant. Another possible use of AEPL is during the design phase of a new application language: AEPL provides a rapid and cheap way to experiment with different versions of a proposed language.

We believe that the major innovations present in AEPL are the treatment of sets, used to create data structures and to define new data types, and the use of a powerful syntax description mechanism derived from the Markov Algorithm. We think also that most of the power of the system stems from its particular architecture and the concept of a special machine or processor which embodies the semantics of the language.

In what follows, we give a description of the data structure concepts of AEPL and we show how these concepts are used to create complex data structures and new types of data elements. A complete description of the language can be found elsewhere.[15]

The next section of this article presents some of the design objectives of AEPL; the following one describes in general terms the overall model of the AEPL system. Finally, the last section discusses the data structure concepts and the semantics of the data definition facility.

* Based in part on a thesis submitted by the first author to the Faculty of Electrical Engineering of the Technion in partial fulfillment of the requirements for the degree of Doctor of Science in Technology.

** Presently with the Department of Applied Mathematics, University of Louvain, 3030 Heverlee, Belgium.

*** On sabbatical leave at the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California 94720.

GENERAL DESIGN OBJECTIVES

During the design phase of AEPL, we tried to remain consistent with a number of general concepts and ideas which we discuss in this section.

Extensibility

The three main aspects of extensibility which we set out to provide were the ability to define new types of data items and new operations on old or new data items and the possibility to modify extensively the syntactic frame of the language. The AEPL system was designed so as to present itself to the users as a language, sometimes called *core* or *kernel* language, which includes a number of basic data types, a number of operators for these data types and a syntactic frame within which one can describe sequences of operations on data, i.e., programs. The core language includes also the tools which enable one to modify these basic constituents and create 'extended languages'. Note, however, that the adjective 'extended' does not necessarily imply addition of features to the core language: one can use the extension mechanisms to produce a language which is less rich than the kernel by deletion of undesired features.

Minimality

In the design of an extensible language, one is tempted to limit the number of primitive language features to the bare minimum and to rely on extensibility for the creation of useful languages from the original core language. While the precise definition of a minimum set of features is a problem in itself, it is clear that the emphasis on minimality leads to kernel languages which are so primitive and involuted that their use is difficult: they have to be drastically extended in order to be of any practical use.

The design of AEPL is a compromise between a desire to keep the number of features of the kernel as low as possible and the requirement that the language be a fairly convenient programming tool.

Generality and completeness

Rather than to emphasize minimality, our approach has been to try to limit the number of primitive concepts, not the number of built-in language features. For that purpose, we tried to isolate a few very general ideas regarding data structures and syntax and to implement them in a language which would respect the concept of completeness as expressed by Reynolds:[17] any value or class of values which is permitted in some context of the language should be permissible in any other meaningful context. This makes the langauge very regular: the number of special cases and particular conventions is greatly reduced. We believe that this is an important feature for an extensible language, since it reduces the number of possible inadvertent violations of the language rules.

THE MODEL

The AEPL system is composed of three parts:

- a core language,
- a processor,
- a translator.

1. The AEPL core language is a relatively small language which resembles Algol 60 in the sense that it includes a number of basic expression and statement forms (including declarations) and that the name-scoping of its variables is governed by an Algol-like block-structure. It differs from Algol 60 in the following aspects:

    - the primitive data items manipulated in AEPL are not the integer numbers, real numbers, arrays, etc., of Algol 60, but so-called *t*-values and objects as described below;
    - the AEPL core language contains a data definition facility which enables the user to define and manipulate new data structures;
    - the AEPL core language includes a number of facilities for modifying its own translator, thereby allowing an extensive syntactic variability.

2. The AEPL processor is a machine which operates on data structures of a particular kind, namely executable data structures called programs. Programs may be created and operated upon by the user in the same way as any other data structures. Programs are distinguished only by the fact that if the AEPL processor is applied upon them or, more precisely, if control is transferred to a program data structure, a number of actions will be performed by the processor.

    The AEPL processor recognizes 63 different kinds of programs, i.e., the processor is a machine with a repertoire of 63 different instructions. It is possible to combine a number of such programs into a compound data structure; control can then be transferred to this structure and the processor will then execute the different actions specified by the individual programs in a well-defined order.

3. The AEPL translator is a program for the AEPL processor whose purpose is to transform an input string of characters into another data structure according to the rules of a special kind of grammar. At certain points of the translation, control may be transferred from the translator program to certain parts of the generated structure, thereby yielding "execution" of the transformed text by the processor.

   The AEPL translator is composed of a lexical scan and a parsing phase. The parser consists of a parsing algorithm derived from the Markov Algorithm[8,11–13] and a modifiable grammar which 'drives' the algorithm. The source text submitted by a user may contain statements whose execution affects the grammar by addition or deletion of rules. This feature is used to modify the syntax of the language: one may add new operators, new kinds of expressions, new types of statements dynamically; it is also possible to redefine (overload) or delete existing language structures.
4. In conclusion, one may view the AEPL system as consisting of a program (the translator) executed on a special machine (the processor). The translator transforms the input into several data structures. A certain number of those data structures can be interpreted as instructions for the processor and control can be transferred to them. If the input contains the appropriate command, the execution of the corresponding data structures by the processor will modify the translator: the language will have been extended.

   The translator program is present in the memory of the processor together with the generated data structures unless those have been deleted by specific commands. Thus, at any instant of time, the "run time environment" of a user's program consists of the whole AEPL system augmented by the programs which were executed in the past and the data structures resulting from the execution of those programs. This approach is similar to that of languages such as LISP and BALM.[9]

   Since the processor is implemented conceptually as a program executed on an existing computer, the AEPL system can be considered to be interpretative.

DATA STRUCTURES

Principles

One of our aims has been to create in AEPL a simple but general data definition and manipulation facility which would allow us to handle a wide class of data structures. This facility should be powerful in order to enable the user to define complex data organizations; it should however be simple enough to understand and to use. This last point required that the data structure facility be based on a small number of well-chosen primitives.

Another design decision which has been made regarding AEPL is the total separation between data structures as conceptual organizations of data and storage structures or representations of data structures in memory. At present, the user is provided with a flexible data structure manipulation system, but he has no control over the way the structures are represented in memory.

It is clear that an algorithm can be specified and checked out for logical flaws without reference to memory representations. Indeed, when a complex algorithm is designed, it is common practice to clear the main issues and to avoid excessive detail by specifying the data structures first and postponing decisions regarding memory structures to a later stage. On the other hand, it is certain that the efficiency of any algorithm depends on the memory representations of the data structures. Therefore, in its current form, AEPL is a tool which is useful in the first stage of the design of algorithms. Using AEPL, one can verify and debug an algorithm in terms of its logic rather than in terms of its storage structures. After the debugging phase, however, it may be necessary to modify the default storage structures in order to increase the efficiency of the algorithm. At this stage, it is certainly easier to experiment with new storage structures, since one is at least almost certain that the *logic* of the algorithm is correct.

A complete programming system such as the one we aimed at should also provide means for controlling and checking the memory representations. This requires an *implementation specification language* which would allow the specification of storage structures by addition of statements to a program rather than by the modification of the program. This idea is not new: it has been proposed by Balzer,[1] Schwartz,[2] and Earley[6] among others.

Basic data elements

There are two kinds of data elements in AEPL: *t-values* (terminal values) and *objects*. Both kinds are strongly interrelated.

—*t*-values are entities which can be used as values (in the sense described below) of attributes of objects. Examples of *t*-values are integer numbers, character-strings, sets of integer numbers.

—Objects are entities to which six *t*-values are associated in the following way: we say that an object possesses six attributes, named respectively:
name-, value-, mode-, type-, scope-, and
rule-attribute.

Each attribute may possess a value, which is necessarily a *t*-value. If an attribute of an object possesses no value at some point in time, it is said that its value is undefined. It is possible to inquire about the value of any attribute of any object, and to modify that value.

Another way of looking at this would be to say that one object describes particular relationships between the six *t*-values which are the values of its attributes. The nature of these relationships will be explained below.

T-values

The AEPL system provides the following kinds of *t*-values:

—atomic *t*-values: integers, reals, character-strings, labels and references;
—compound *t*-values or, in our terminology, sets:
—explicit sets or E-sets,
—conceptual sets: C-sets, R-sets, P-sets, U-sets, I-sets, F-sets and primitive sets.

The primitive sets are:

—the set of all integer *t*-values,
—the set of all real *t*-values,
—the set of all character-string *t*-values,
—the set of all label *t*-values,
—the set of all reference *t*-values.

Although the term "set" is used, the concept is not in every case the same as the one used in mathematics. Some of the AEPL sets are ordered and may contain the same element many times; other sets (e.g., the primitive sets) correspond precisely to the mathematical notion of set: an unordered collection of distinct elements.

Classes of *t*-values

The set of all integer *t*-values is sometimes called the *class* of all integer *t*-values; similarly, the other primitive sets are primitive classes. The term "class" is used for a set which specifies the "kind" or "type" (in the Algol 60 sense) of a *t*-value. The primitive classes are available in the kernel language; other classes can be formed by means of the extension facilities. In fact, any set can be used in AEPL to define a class of *t*-values (see below).

Among the five primitive classes, only the class of reference *t*-values needs further explanation.

References

A reference is a *t*-value which designates an object in a unique way. One of the ways to gain access to the attributes of an object is by using a reference to that object. We do not concern ourselves with the implementation of such references: the important fact is that for every reference *t*-value there exists one and only one object which is referred to by that *t*-value. The reference concept is a generalization of the pointer concept which does not imply any particular implementation.

Sets

As mentioned above, a set, in AEPL, is a collection of *t*-values which is itself a *t*-value. Sets are used:

—to create aggregates of *t*-values,
—to define new classes of *t*-values.

AEPL distinguishes between two kinds of sets: *explicit sets* and *conceptual sets.*

An *explicit set* is a finite ordered collection of *t*-values which are effectively present in the system. Such sets correspond to the usual programming concepts of vector, list or sequence. An example is the explicit set composed of the integer *t*-values one, two and three, in that order.

A *conceptual set* is a collection of *t*-values which is defined implicitly. It may be finite or infinite, ordered or unordered. Such a set is defined by a predicate: it consists of all the *t*-values for which the predicate is true. In mathematical notation:

$$\{x \mid P(x)\}$$

An example is the set of all integer *t*-values, or the set of all character strings beginning with the letter A, or the set of all prime numbers smaller than 100. Sets are described in greater detail below.

Other classes of *t*-values

There are a number of classes of *t*-values which are not primitive classes, but which are used within the translator program. Because of the model described above, the data structures of the translator are accessible to the user. Among other structures, the translator for the core AEPL uses a number of classes, called *built-in* classes, which define domains of *t*-values which may be of interest to the user: these classes are built in terms of the primitive classes in the same way that user-defined classes are constructed. Among these built-in classes is the class of all identifier *t*-values (character-strings beginning with a letter and containing only letters or digits) and the class of program *t*-values (explicit sets which can be interpreted as commands to the AEPL processor).

Objects and their attributes

Objects are entities to which six *t*-values are associated: one object describes specific relationships between these *t*-values, which are said to be the values of the *attributes* of that object. We describe here the roles of the attributes of an object.

The attributes of an object

The name-attribute

The value of the name-attribute of an object or, for short, the name of an object, is a *t*-value belonging to the class of identifiers: it may be used to refer to an object in the same way as a reference *t*-value. An identifier is thus associated with an object through the name-attribute of that object. Many objects may have the same identifiers as value of their name-attribute, but at every point in time a given identifier may be used to refer to only *one* of these objects. The choice of the object which is referred to by a given identifier is governed by the name scoping rules which depend on the block structure of the text submitted to the translator.

The value-attribute

The value of the value-attribute of an object or, again for short, the value of an object is a *t*-value whose class is defined by the mode-attribute of the object (see below). This attribute is closely related to the usual concept of value of a constant or of a variable in other programming languages.

The mode-attribute

The value of the mode-attribute of an object α is a reference to an object β whose value is a set of *t*-values to which the value of α belongs. The value of β thus defines the domain of the values of α or their class. For short, we say that β *is* the mode of α or that object α *possesses* mode β.

The reason for the existence of the mode-attribute is simply to allow the association of a meaning with the internal representation of the value of an object. The mode of an object α will indeed indicate whether the value of α is an integer *t*-value, a reference, a set, and so on. The corresponding Algol 60 concept is that of type of a variable; the name "mode" has been chosen because of the similarity with the Algol 68[22] idea. Another purpose for the mode-attribute is its use, similar to that of *syntactic type*, during the parsing process. The modes of the objects involved in the parsing play indeed an important role in the selection of the grammar rules which must be applied to transform the input string into the parse tree.

The type-attribute

The type-attribute of an object can possess two values which indicate whether the object is a *variable* or a *constant*. An object is variable if the set of possible values for that object contains more than one element; otherwise it is constant. Clearly, one could indicate that an object α is constant by having its mode be a reference to an object β whose value is a set with one element, namely the value of α. However, it is usually preferable not to use this device; it is more appropriate to distinguish between a variable object whose value is the integer *t*-value seven and a constant object whose value is the integer seven by means of the type-attribute. The mode-attributes of these two objects could then both be a reference to an object whose value is the set of all integer *t*-values.

The scope-attribute

The scope-attribute of an object can possess three values denoted GLOBAL, LOCAL and DUMMY which define the scope of the relationship between the object and the identifier which is the value of its name-attribute.[15,16]

The rule-attribute

The purpose of this attribute is related to the generation of program *t*-values by the parsing process.[15,16] The value of the rule-attribute belongs to the built-in class of program *t*-values.

Primitive objects

To all primitive classes correspond built-in primitive objects. We thus have an object whose name is INT and whose value is the class of all integer *t*-values. The mode of this object has to indicate that its value is a primitive class: this is achieved by having the mode of object INT be a reference to a special object known to the system as the object whose name is PRIMITIVE and whose value is the set of all primitive classes. The value of the mode of PRIMITIVE is undefined. (The program which operates on the data structures recognizes the name PRIMITIVE.)

An example

Figure 1 illustrates, through a schematic representation, the relationships among three objects. The object A, i.e., the object whose name-attribute has the identifier A as value, has as other attributes:

—the value is the integer *t*-value twenty-seven (the dotted line is used to indicate this),
—the mode is a reference to the object INT,
—the type is the *t*-value indicating that A is a variable,
—the scope is the *t*-value indicating that the association of the identifier A with this object is global,
—the rule is irrelevant (its value is either undefined or not important in this context).

Sets—detailed description

Explicit sets

An explicit set or *E-set* is a finite ordered collection of *t*-values. It corresponds to the usual notions of vector, list or sequence. Every member of such a collection is called a *component.* E-sets may be used to create aggregates of data or to define new classes of *t*-values.

The basic operations on E-sets are:

—selection of a component by ordinal position or by name (retrieval or storage of a value),
—addition or removal of a component,
—selection of a subset,
—test for membership,
—finding the number of components,
—concatenation of two E-sets.

The language possesses a notation for constant E-sets, e.g.,

E{1,2,'ABC,'E{3,4}}

which denotes an E-set of four components, the fourth of which is itself an E-set with two components.

One can define other operations on E-sets in terms of these basic operations by means of the syntactic extensibility mechanism of the language.

If one wishes, for instance, to introduce *unordered finite* sets of *distinct* elements, one can do so easily by representing these sets as E-sets and by ignoring the ordering relation among the components. At least two basic operations must however be redefined:

—the test for equality between two sets must ignore the ordering,
—the addition of an element to a set must verify that the element is not yet a member of that set.

Other operations on unordered sets (union, intersection, power set, and so on) can then be written in terms of the basic operations. Unordered sets of ordered pairs may be used, as in SETL,[20] to represent mappings; functional application can then be easily defined for such mappings.

E-sets can be used to define new classes of *t*-values by enumeration. For instance, the set E{'1','3','5', '7','9'} could be used to define the class of odd digits. Similarly, the set E{1,2,3,5,7,11,13} could be used to define the class of prime integers smaller than 15.

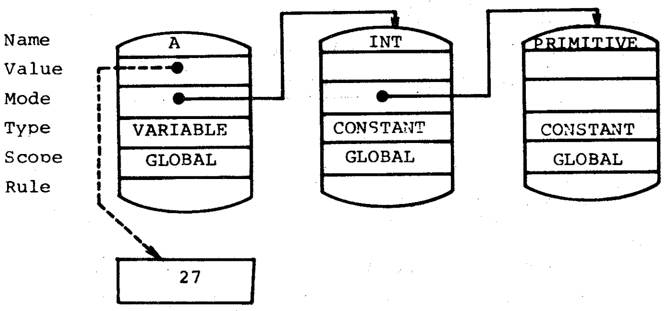


Figure 1—The objects A, INT and PRIMITIVE

Conceptual sets

A conceptual set is a set defined by a predicate. Such a set is not present in the system under the form of a collection of *t*-values: it is present purely by convention as the set (in the mathematical sense) of *t*-values for which the predicate is true. A conceptual set is thus a collection of *t*-values defined by a certain common property. These sets are represented in AEPL by *descriptions* of the properties of their elements rather than by a list of their elements. Since such a description is usually composed of several elements, AEPL represents a description by an E-set. The description of a conceptual set may be stored in the value-attribute of an object; the mode of that object will indicate that its value may be interpreted as the description of a conceptual set.

The primitive sets are conceptual sets corresponding to the primitive classes of AEPL: they exist in the system as the values of the primitive objects INT, REAL, CHAR, LABEL and REFERENCE. Other conceptual sets belong to one of the following categories:

C-sets, R-sets, P-sets, U-sets, I-sets and F-sets.

The reason why there is more than one kind of conceptual set besides the primitive sets is simply one of ease of programming: it is not always convenient to represent a set by a general predicate; certain particular cases deserve special treatment.

The basic operation involving conceptual sets is the test for membership.

C-sets

According to the functions of the attributes described above, if an object α has an E-set as value (i.e., as value of its value-attribute), then the mode of α should be a reference to an object β whose value is a set of E-sets, namely the class to which the value of α belongs. This class may be defined by a C-set: a C-set is indeed a set of E-sets. Its description is composed of the following five *t*-values:

Number-type is a *t*-value which indicates whether the number of components of the E-sets which belong to this C-set is variable or constant.

Number is a *t*-value which is the number of components of the E-sets which belong to this C-set if this number is constant (examine number-type to find this out); otherwise, this *t*-value is a reference to a Boolean function of two arguments: an integer *t*-value n and a reference to an object α whose value belongs to the class of E-sets described by this C-set. The function returns the value true if and only if the integer n is a permitted value for the number of components of the value of α.

Component-type is a *t*-value which indicates whether the E-sets belonging to this C-set are *homogeneous* or not. A homogeneous E-set is one whose components belong to the same class.

Component-class is a *t*-value which defines the class of every component of any E-set belonging to this C-set in the following way. If the E-sets are homogeneous (examine component-type to find this out), then component-class is a reference to an object whose value is the class to which all the components belong. If the E-sets are nonhomogeneous, then this *t*-value is a reference to a function of two arguments: an integer *t*-value n and a reference to an object α whose value belongs to the class of E-sets described by this C-set. The result of this function is a reference to an object whose value is the class to which the nth component of the value of α belongs.

Names is either undefined or a reference to a function of two arguments, an identifier *id* and a reference to an object α whose value belongs to the class of E-sets described by this C-set. The function returns an integer *t*-value n which is the ordinal position of the component of the value of α whose name is to be *id*. If no such component is found, then the function returns zero.

Figure 2 schematizes an example in which the object PAIR has, as its value, the set of all E-sets with two integer components which are unnamed. The values of number-type (constant), number (2), component-type (constant), component-class (a reference to INT) and of names (undefined) define this by convention. The value of object X (a pair of integers) belongs to the class defined by PAIR, so the mode of X is a refer-

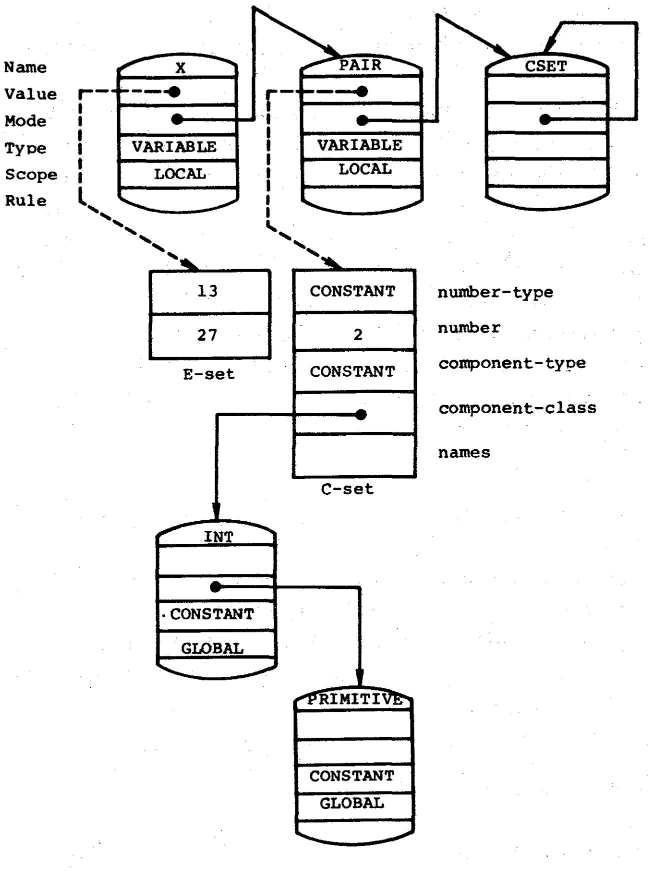


Figure 2—A possible data structure for an object X whose value is a pair of integers

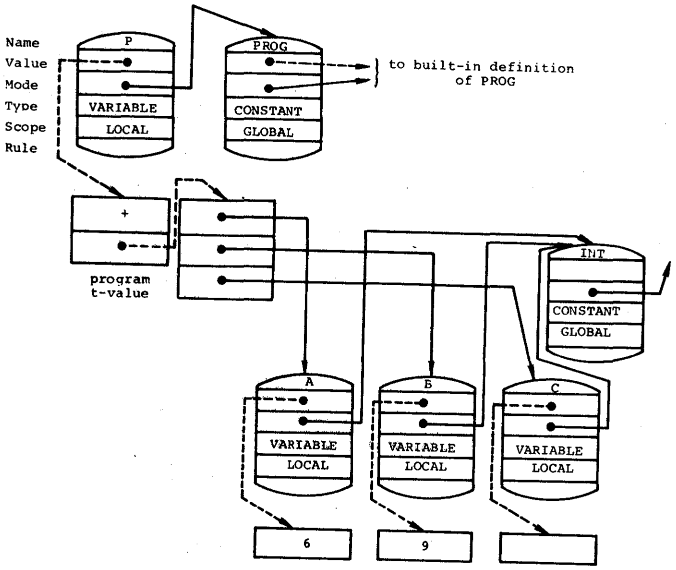


Figure 3—A program *t*-value and its structure

ence to PAIR. We wish to point out there that the schematic representation of an E-set as shown in Figure 2 does *not* imply any particular implementation.

Another example of a class of E-sets can be found in Figure 3 which illustrates the structure of program *t*-values.

A program *t*-value is a particular kind of E-set which can be interpreted by the AEPL processor as a command to perform some actions. The class of program *t*-values is predefined in the AEPL system;[16] the example of Figure 3 shows a program *t*-value composed of two components: an operator and an E-set of three references to objects A, B and C. The "execution" of this structure by the AEPL processor will cause the sum of the values of objects A and B (namely the integer *t*-value 15) to be stored as the value of the value-attribute of object C.

Other conceptual sets

In order to shorten this presentation, we shall not give the complete descriptions of the components of the other kinds of conceptual sets here. We shall limit ourselves to an informal description of the properties of the conceptual sets.

R-sets (restriction sets)

R-sets are used to impose a restriction on the elements of another set. This restriction takes the form of a Boolean function which specifies which *t*-values are members of the restricted set. In mathematical notation:

$$\{x\epsilon S \mid P(x)\}$$

Examples of sets defined by means of R-sets:

—the set of all positive integers,
—the set of all prime integers,
—the set of all prime integers smaller than 15,
—the set of all character-strings beginning with the letter A.

P-sets (property sets)

This kind of conceptual set is similar to the class of R-sets; it defines a subset S′ of a given set S by distinguishing a specific property. This property need not, however, be expressed as a predicate; the user must nevertheless specify how the property is to be used to distinguish the elements of the subset. This involves modifying the membership operation to perform the appropriate actions when testing for membership in S′. P-sets are thus an escape mechanism enabling the user to design different kinds of conceptual sets.

U-sets and I-sets (union and intersection sets)

These conceptual sets make it possible to define classes of *t*-values as unions or intersections of other sets. Examples of sets defined in such a way:

—the set of all integer and real *t*-values,
—the set of all prime integers smaller than 15.

F-sets (file sets)

These sets are used to define input-output sequential files.

CONCLUSION

In this article, we have presented an overview of the main features of the AEPL system. We have discussed in detail the data structure concepts which form the basis for the data definition facility of AEPL. Using these concepts, a user can create, in a straightforward manner, new kinds of data items and aggregates of data. Other features of the system enable the user to define new operators for any kind of data items and to create new language structures such as statement forms.

Our experience with the language has been limited to pen-and-paper coding since the system is not yet implemented. A language for the creation and manipulation of linear graphs obtained by extension of the kernel AEPL is described in detail elsewhere.[15] In our opinion, this and other examples show the feasibility and the usefulness of the approach described in this paper.

ACKNOWLEDGMENTS

We would like to express our gratitude to the following people for the numerous and fruitful discussions we had with them: Prof. Y. Wallach and Dr. E. Kantorowitz of the Technion, Prof. J. Feldman of Stanford University, Profs. J. C. Boussard and M. Griffiths of the University of Grenoble and Messrs. S. Schuman and P. Jorrand of the IBM Scientific Center in Grenoble.

REFERENCES

1 R M BALZER
Dataless programming
Proc AFIPS 1967 FJCC pp 535-544

2 J R BELL
The design of a minimal expandable computer language
Doctoral dissertation Stanford University 1968

3 T E CHEATHAM Jr
The introduction of definitional facilities into higher level languages
Proc AFIPS 1966 FJCC pp 623-637

4 T E CHEATHAM Jr A FISHER P JORRAND
On the basis for ELF—An Extensible Language Facility
Proc AFIPS 1968 FJCC pp 937-948

5 C CHRISTENSEN C J SHAW (eds)
Proceedings of the Extensible Language Symposium
SIGPLAN Notices 4 (Aug 1969) pp 1–62

6 J EARLEY
Toward an understanding of data structures
Comm ACM 14 10 (Oct 1971) pp 617-627

7 B A GALLER A J PERLIS
A proposal for definitions in Algol
Comm ACM 10 4 (April 1967) pp 204–219

8 B A GALLER A J PERLIS
A view of programming languages
Addison-Wesley Publ Co Reading Mass 1970

9 M C HARRISON
BALM—An extendable list-processing language
Proc AFIPS 1970 SJCC pp 507-511

10 E T IRONS
Experience with an extensible language
Comm ACM 13 1 (Jan 1970) pp 31-40

11 J KATZENELSON
The Modified Markov Algorithm as a language parser—linear bounds
J of Systems and Computer Sciences to be published

12 J KATZENELSON E MILGROM
The Markov Algorithm as a language parser
In preparation

13 A A MARKOV
Theory of algorithms
Academy of Sciences of the USSR 1954 English translation by Israel Program for Scientific translations

14 M D McILROY
Macroinstruction extension of compiler languages
Comm ACM 3 4 (April 1960) pp 214-220

15 E MILGROM
Design of an extensible progammming language
Doctoral dissertation Technion—Israel Institute of Technology 1971

16 E MILGROM J KATZENELSON
AEPL—An Extensible Programming Language
In preparation

17 J C REYNOLDS
GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept
Comm ACM 13 5 (May 1970) pp 308-318

18 J E SAMMET
Programming languages: history and fundamentals
Prentice-Hall Co Englewood Cliffs N J 1969

19 S A SCHUMAN (ed)
Proceedings of the International Symposium on Extensible Programming Languages Grenoble 1971
SIGPLAN Notices 6 12 (Dec 1971)

20 J T SCHWARTZ
Abstract algorithms and a set-theoretic language for their expression
Preliminary draft Courant Institute of Mathematical Sciences New York University New York N Y 1970-72

21 N SOLNTSEFF A YEZERSKY
A survey of extensible programming languages
Computer Science Tech Report No 71/7
MacMaster University Hamilton Ontario 1971

22 A VAN WIJNGAARDEN (ed)
Report on the Algorithmic Language ALGOL 68
Numerische Mathematik 14 (1969) pp 79-218

The universal consulting language alias—The investment analysis language

by CAROLE A. DMYTRYSHAK

Bankers Trust Company
New York, New York

INTRODUCTION

IAL (Investment Analysis Language) is a computer language which can be used to generate economic forecasts, develop data bases with complicated list structures, analyze results from psychological tests or compare alternative investments. Unfortunately, the label "Investment Analysis Language" has really limited the number of people who have considered using the language. IAL has been developed as a tool for a group of internal consultants in order that they can solve these problems quickly taking into consideration the types of problems they are asked to solve, the tools available and their own talents. Unfortunately, since IAL was developed for a bank, the label "investment analysis" was used in its name. A more appropriate title would be the UCL, Universal Consulting Language. Regardless of the industry, regressions, statistics and adaptive forecasting are performed in the same manner.

I am convinced that the use of the Investment Analysis Language is a good way of getting the maximum output from a group of internal consultants (i.e., Management Science, Financial Analysis or Operations Research Group) in the shortest period of time. To illustrate my point, I will present the history of IAL, discuss the need for developing such a language, explain the characteristics of IAL and why they are essential to an efficient operation. A specific example of the use of IAL at Bankers Trust, the support the language receives from the American Bankers Association and future uses of the language, will clarify the major arguments.

THE NEED FOR IAL

The problems

Most corporations concentrate a collection of bright young men and women in departments such as Management Science, Operations Research, Financial Analysis and Planning or Corporate Planning. Regardless of the formal title, the composition of the group and the problems it is asked to solve are of the same structure. The group is composed of highly paid people who have advanced degrees in business administration, operations research, management science, economics or some other related field.

The problems they are presented with generally have the same set of characteristics:

1. They must be solved quickly. For example, an analysis to look at the effect of a change in the prime lending rate must be completed and all the financial implications reported to the chief lending officer of a bank before he gets impatient and makes a decision without the benefit of the knowledge gained from the study. Because of this need for faster solutions, time-shared computer systems are one of the tools used by the group.
2. A request to solve the problem is often a one-time assignment. For instance, a study on leasing 747's to an airline is needed only once then the leasing group is out looking for new clients.
3. When performing the analysis, many factors must be taken into consideration and these factors tend to make the problem highly technical and complicated. When, looking at a leasing problem, we must take into consideration tax rates, depreciation schedules, reinvestment rates and the resale value of the equipment. It is impossible for even the most skilled analyst to take all these factors into account without the use of a computer.
4. The programs developed for the analysis must be flexible. Often, after working on a problem, the results will lead to other questions. To answer these questions another degree of so-

phistication must be added to the program and quickly.

5. Similar types of analysis require the same basic operations to be performed on different data. To analyze financial deals one always performs future value or present value calculations.

As can be seen by these characteristics, it is essential that problems be solved quickly. Yet these very characteristics make it twice as hard to solve the problem. The organization of the group and the tools used by the group are key factors in achieving fast throughput.

Previous solutions

There are three basic structures which have been followed in organizing a group of internal consultants. First, and most expensive, is an all-analyst staff attempting to do their own programming as well as analysis. This is expensive both in terms of salaries and in utilization of talent: analysts are generally paid more than programmers but are not efficient programmers. Nor are analysts interested in investing much of their time in extensive programming tasks. The other end of the spectrum is an all programmer staff to perform analysis as well as programming. The only point in favor of this method is that programmers are not as highly paid as analysts. Programmers are often pressured into giving the user just what he requested in terms of printouts but in most cases are not permitted to make the in-depth analysis required to solve the problem.

The middle of the road, and most common approach, is to have programmers and analysts work together on the same problem. This method does work and appears very efficient at first glance. However, it breaks down because the analysts and programmers communicate and operate on two distinct levels. Communication is always a problem whenever professionals from two diverse fields are assigned to the same projects.

When the analyst and programmer finally do find a basis for communication, they may be creating an unnecessarily complex and unique solution. The analyst's main concern is structuring the analysis to cover any contingency. He knows that he is working with an expert who has mastered the computer, and, therefore, he has no hesitation in requesting revisions to a program and adding another degree of complication to the study with very little consideration for the marginal cost of the revision. If he were told, "We can make those changes but it will take three days of reprogramming" he might not find the revisions quite as necessary.

On the other hand, the programmer is anxious to please and may not object to adding another degree of complexity to an already complex problem. In most cases, the end result is a monster program that has taken several weeks or months to develop and is not general enough to handle anything but the immediate problem.

A new solution

In 1968, David M. Ahlers became the head of the Management Science Group at Bankers Trust (BTCO) and he brought with him the basic modules for the IAL system. The idea and the modules for the language had been developed over a period of several years while Mr. Ahlers had been studying and teaching at Carnegie-Mellon University.

Mr. Ahlers wanted to be able to teach financial concepts as applied to real problems without falling into the trap of teaching programming. He needed a computer tool that was easy to understand and use. When consulting, he could not afford to start from scratch and program every phase of an assignment. In order to save time in future projects he began to save the routines he had written for performing those calculations which were common to most projects. Since he had to work on several different types of computers his routines had to be easily transferred and basically, hardware independent. These needs led to the development of IAL. Since the problems faced when teaching and consulting are very similar to those of Bankers Trust's Management Science Division it was decided to use IAL for most projects. Revisions were made to the language and it was installed on three different time-sharing systems used by Bankers Trust. The user's manual was completed and later the language leased to The American Bankers Association for distribution to commercial time-sharing vendors.

DESCRIPTION OF IAL

IAL is a computer-based language consisting of over 60 functions which can be called on to perform analysis in areas ranging from time value of money to adaptive forecasting. The language is structured so that users of different degrees of proficiency in any area are able to use it and in some cases to become quite sophisticated with increased use of IAL.

In the following discussion the hypothetical user is never termed "programmer." The principal user should be an analyst with specialization in business administration, economics, finance or some other related field. IAL has been designed in such a way that the user does not have to know how to program, nor spend a great deal of time learning how to use an interactive terminal.

The analyst will find it faster to specify his needs directly, using IAL, rather than calling on the services of a programmer. Because of this tool the structure of the internal consulting group will naturally evolve into a group of analysts and perhaps one or two programmers. These programmers will be assigned projects requiring expertise in the computer field.

Based on FORTRAN

IAL is written in a subset of FORTRAN. This set of FORTRAN is composed of the intersection of the FORTRAN languages available on commercial time-sharing when IAL was developed. In its present form IAL is very similar to FORTRAN II and contains the standard operations of that language. The primary aim was to be able to install the language quickly, without any major modifications, on any time-sharing system which offered FORTRAN. IAL is now provided as a service by six different time-sharing vendors. It is running on computers made by five different manufacturers. Installation has not presented many problems to date.

Blocks of commands

IAL is composed of a set of blocks and functions within those blocks. The blocks can be combined in any fashion. The functions within each block are related by the kinds of jobs they perform. For example, the time value of money (TIMEE) block contains all of the future value and present value commands, the teletype input/output (ENTEL) block contains all of the print and read commands for the teletype, and the graphing routines are in the block ENGRF. The language was divided into blocks for two reasons; the time-sharing system at that time did not have large quantities of core available for the user and it was felt that a user would be able to gain a better understanding of the commands if they were organized by functional area. Table I contains a list of the blocks and their functional areas.

TABLE I—Blocks of Commands

Block Name	Functional Area	Number of Commands
ENTEL	Input/Output of data via a terminal,	8
QTRNK	Utility functions for statistical routines	2
ENFIL	Save and retrieve data from permanent storage devices	5
ENGRF	Graphing data	2
TDATA	Transformation of data	1
TIMEE	Time value of money calculations	15
INTRR	Internal rate of return computations	3
DEPRE	Depreciation and tax credit analysis	4
QALTS	Evaluation of qualitative time series (1 or 2 series)	5
QALCP	Evaluation of qualitatives time series (continuation of QALTS)	4
CAPBD	Capital budgeting and risk analysis	3
NUMTS	Forecasting of numerical time series	4
REGRS	Correlation and regression analysis	4
INVRS	Inversion of matrices	1

Each command—a function

Each command in IAL is a FORTRAN function and the characteristics of the FORTRAN functions have been incorporated into the language. The communications link between the main program and the various functions is the function name and the argument list. The function name is used for program control while the argument list is for data flow. As an example, command RELAT was designed to determine if two data series are statistically related and if so what is the relationship. If the user were looking at series A and B each with N points he could execute the following call in his program

```
CALL RELAT (A, B, N).
```

If he wanted to take advantage of the control parameter passed through the function name he could nest RELAT in QUEST.

```
CALL QUEST (1, RELAT (A, B, N)).
```

QUEST will translate these control parameters and display one of the following messages depending upon the relationship between the two series.

```
THE ANSWER TO QUESTION 1 IS NO
THE ANSWER TO QUESTION 1 IS YES
  THE 1ST SERIES IS GREATER THAN THE
    2ND
THE ANSWER TO QUESTION 1 IS YES
  THE 2ND SERIES IS GREATER THAN THE
    1ST
```

The command structure gives two advantages:

(a) The user has complete control of the information that is displayed. In many computational packages if the user calls the regression command, the system will print out the coefficients, the intercept, the standard deviation, the residuals, the Durban-Watson coefficient and any

TABLE II—I/O Commands

Name	Function	Call Statement*
TTY1	Enter 1 value via terminal	TTY1 (ID, X)
OUT1	Print 1 value on the terminal	OUT1 (ID, X)
INTTY	Enter table, Y, with N rows and M columns	INTTY (ID, Y, N, M)
NOUTT	Print table Y with N rows and M columns	NOUTT (ID, Y, N, M)
SAVE	Store on a permanent storage device table Y with N rows and M columns	SAVE (ID, IF, Y, N, M)
NFETCH	Retrieve from a permanent storage device table Y with N rows and M columns located on line J	NFETCH (R, IF, Y, M*N)
QUEST	Print out the results of statistical tests in terms the user can understand	QUEST (IF, FN)

* In the command calls
ID—the output identification number
X—variable being displayed or entered
Y—table being displayed or entered
N—number of rows
M—number of columns
IF—file number
FN—name of statistical function
R—location where retrieving is to begin

other statistic that captured the programmer's interest at the time the package was developed. This is wasteful since not all users understand how to correctly interpret the results. When using the regression command in IAL all of the relevant statistics are calculated and passed to the main program through the argument list. The user then has the option to print any of the values he feels are needed for the analysis.

(b) The concentration of the I/O routines in a limited set of commands has made installation of IAL on various time-sharing systems relatively easy. It has been found that the execution of the I/O commands are the most system-dependent operations on any time-sharing system. When installing IAL on a new system, the nine I/O commands have to be modified but the computational routines are transferred without further modifications. Flexibility was a key ingredient to early users of time-sharing systems since they were required to make frequent switches in services, for one reason or another.

User works at own level

IAL always allows the user to work at his own level of sophistication—to the extent that he feels at home with the language. For example, a user can call one command (SFORM) to look at a time-series to determine what the forecast for next time period is. This command will provide him with the results and these results will be dependent upon whether he indicates that the series has trend and seasonal components. If the user does not know or can not make an assumption as to whether the trend or seasonal components exist, he can include in the argument list calls to the TREND and SONRA commands.

```
CALL SFORM (A,12,TREND(A,12),SONRA(A,
12),B)
```

The commands will then set the necessary parameters for the SFORM command using statistical tests based on a 90 percent confidence interval. If the user is more knowledgeable in statistical analysis and would like to set up a different set of confidence limits or to know the underlying statistics used in determining the results, he is able to run the SFORM command and set up the necessary parameters

```
CALL INTTY (1,A,12,1)
CALL SFORM (A,12,1.,1.,B)
```

He can also descend to another level and use the regression command which is basic to SFORM.

The use of the I/O commands varies heavily depending upon the level of the sophistication of the user. The user has the option of using the commands to store individual lists of data in files or to create data structures of linked lists. For instance, in the reporting system for BTCo's Corporate Planning model, a directory system consisting of three levels is used. This directory is used for retrieving data as well as for creating the headings and subheadings used by a new report generator. Because of the careful design of the general purpose file manipulation commands, the user is given a powerful set of tools to handle his storage problems. He can use this system for basic storage of data or build a directory system for the storage and retrieval of the information. At the core of the data storage and retrieval system are the two commands SAVE and NFETCH. Their tasks are explained by their names. SAVE stores data and NFETCH retrieves information stored by the SAVE command.

If the user wishes to store a Table A containing 6 rows and 3 columns in the file designated as File 2, his program would contain the command

```
CALL SAVE (1,2,A,6,3)
```

When the system executes the SAVE command in the user's program, the system will issue the message

SAVE 1 FILE 2 LINE 1001.

The line identification (1001.) tells the user the line number at which the header items for the save and first data element are stored. The line number is used by the NFETCH command for retrieving the data. To retrieve information the user's program will need the following command

```
CALL NFETCH (1001.,2,A,3*6)
```

From these two simple commands a complicated directory system can be developed. To build a simple directory of a series of 6 tables, each containing 2 rows and 5 columns, the user could write the following program:

```
  DO 1 J=1,6
  CALL INTTY (J,A,2,5)
  B(J) =SAVE(J,1,A,2,5)
1 CONTINUE
  CALL SAVE(7,2,B,6,1)
```

The user will have stored list B containing the line numbers of his 6 saves in File 2 and the 6 tables in File 1. The portion of a program which automatically retrieves the third data series is

```
CALL NFETCH (1001.,2,B,6)
CALL NFETCH (B(3),1,A,2*5)
```

The user who does not understand the directory concept or does not have a problem complicated enough to warrant such sophistication is not hampered by any involved system of control codes or options on what to use. To the more sophisticated user the system is open ended, and he may decide upon the level of complexity. The system is also open ended for the novice user, for as his skills grow he too can build more complicated and involved routines.

Providing a language which gives analysts the capability of working at their maximum technical capacity rather than forcing them to work at a level established by a programming package is one of the best ways of getting the greatest value per dollar per analyst.

Open-endedness

The report generating command just added to the system points up another characteristic of the system.

TABLE III—Function Names Related to Job

Function Name	Operation
GRAPH	Produce a graph containing up to 8 data series
PV	Calculate the present value of a number
TREND	Test a data series to determine if a trend exists
RELAT	Determine if two data series are statistically related
GROW	Compute growth rates
UTEST	Perform a Mann-Whitney U Test on a set of data

The language has no apparent limit to the number of commands which can be incorporated into it.

If a user finds that he is performing certain computations frequently, there is no reason why he cannot write a function to perform these computations and use it in conjunction with IAL. At Bankers Trust we have added sets of commands for calculating bond prices, yields and coupons. The IAL user's manual has been designed so that a user can incorporate documentation for the commands he creates. This provides a central library for all routines used by the group.

Function names

The name of each IAL command is related to the task the command performs. Table III shows some of the commands and their related functions. This method of naming functions helps the user analyze his problem and understand the operations he performs, instead of issuing a series of numerical codes as in the case of the early statistical routines. It also provides a means of instant documentation.

EXAMPLES OF THE USE OF IAL

A forecasting problem

One of the most common uses for IAL is in the building of forecasting models and tracking the output of these models.

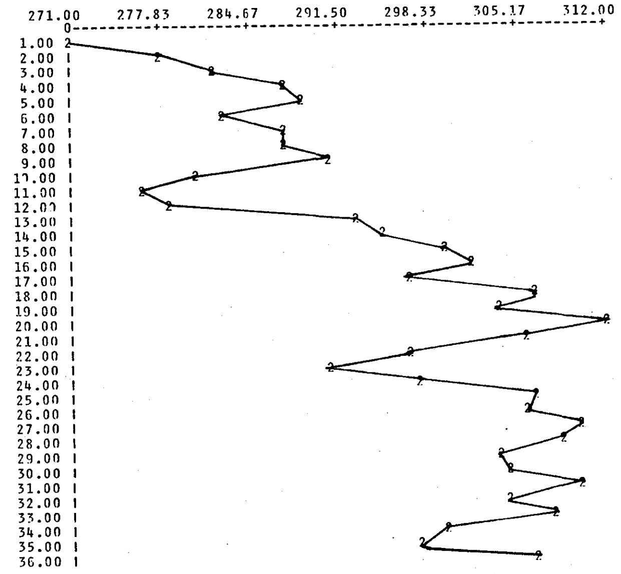
An economist wishes to take the monthly values of an index of business activity for the last 3 years, devise a model to describe past performance and forecast quarterly activity for the next four quarters, and analyze these forecasts.

To do this he has decided to write two programs. The first

will 1 accept the data
2 graph the data
3 devise a model
4 and store the model data on a permanent file

The following program when executed will answer his needs.

```
CALL NFILE(1,'MODEL   ')
CALL INTTY(1,B,0,36)
CALL NGL (B,1,36,2,0)
CALL GRAPH(1,1,6)
CALL OUT1(2,SFORM(B,36,TREND(B,36),
  SONRA(B,36),A))
CALL NOUTT(3,A,10,1)
CALL SAVE(4,1,A,10,1)
STOP
END
```



```
RESULT   2 IS   311.96
RESULT   3
ROW

  1   311.11
  2     0.85
  3     0.0
  4     0.0
  5     2.00
  6     5.38
  7     0.0
  8     0.0
  9   311.96
 10  3799.44
SAVE   4 LINE 1001 FILE  1
```

Graph I

When the program is run the following actions appear on the terminal

```
USING BTC IAL
EXECUTION BEGINS...
  TTY INPUT 1
   36 FREE FORM VALUES
 ?271.,278.,282.,288.,289.,283.,288.,288.
 ?291.,281.,277.,279.,293.,295.,300.,302.
 ?297.,307.,304.,312.,306.,297.,291.,298
 ?307.,306.,310.,309.,304.,305.,310.,305.
 ?308.,300.,298.,307.
```

From the output we see that 311.96 is the forecast for the next month. The equation which describes the behavior of this series is

$$A_t = 311.11 + .85^* t$$

The second program the analyst writes is to revise his original model each month as new data is collected. He will

1 read in the new observation
2 forecast 1 period ahead
3 then forecast 3, 6, 9 and 12 periods ahead
4 Print out the forecasts as well as the upper and lower confidence limits for each forecast

```
CALL NFILE1(1,'MODEL   ')
CALL NFETCH(1001.,1,A,10)
CALL OUT1(2,FORM1(A,1.,TTY1(1),EI))
CALL OUT1(3,FORMT(A,3.,FH1,FL1))
CALL OUT1(4,FH1)
CALL OUT1(5,FL1)
CALL OUT1(6,FORMT(A,6.,FH2,FL2))
CALL OUT1(7,FH2)
CALL OUT1(8,FL2)
CALL OUT1(9,FORMT(A,9.,FH3,FL3))
CALL OUT1(10,FH3)
CALL OUT1(11,FL3)
CALL OUT1(12,FORMT(A,12.,FH4,FL4))
CALL OUT1(13,FH4)
CALL OUT1(14,FL4)
CALL SAVE(1,1,A,10)
STOP
END
```

When the program is run with a new observation of

318 the following results appeared on the terminal:

```
USING BTC IAL
EXECUTION BEGINS. . .
 TTY INPUT  1 IS
?318.
 RESULT  2 IS 314.43←next period forecast
 RESULT  3 IS 316.34←forecast 3 periods ahead
 RESULT  4 IS 323.07↖
                     >confidence limits
 RESULT  5 IS 309.61↙
 RESULT  6 IS 319.21←forecast 6 periods ahead
 RESULT  7 IS 326.99
 RESULT  8 IS 311.42
 RESULT  9 IS 322.07
 RESULT 10 IS 330.93
 RESULT 11 IS 313.22
 RESULT 12 IS 324.94
 RESULT 13 IS 334.89
 RESULT 14 IS 314.99
 SAVE  1 LINE 1041 FILE  1
```

The corporate planning model

The most ambitious undertaking using IAL was the development of the Corporate Planning System at Bankers Trust. The Management Science Division was transferred from the Computer Research and Development Department to the Corporate Planning Task Force in the Office of the Chairman. Our mission was to gather operating data from various sources in the bank, make forecasts and develop a model to be used in the Corporate Planning process. This system had to take into consideration various economic environments, operating characteristics of the bank, possible investment strategies and internal policies. Through this process, a set of expense and income guidelines was developed on a departmental level, covering the next sixteen quarters.

To accomplish this task many people with varied talents were needed. In order to spread the workload, the projects were divided into several small projects with a project coordinator to ensure compatibility of the separate projects. Figure 6 shows how each project fit into the Corporate Planning system. Each project will be discussed separately.

Forecast interest rates and spreads

One of the roles of the Economics Department is to serve as the official forecaster of certain key rates for the bank.

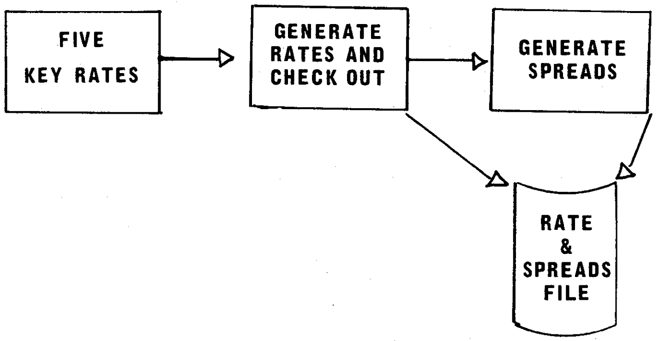


Figure 1—Forecast interest rates and spreads

An analyst was assigned to generate the rates needed to drive the planning model and to act as liaison between the Economics Department and the Task Force. Five key rates were generated by the Economics Department and then the analyst expanded them to provide rates needed for the model. The analyst had to check that these rates were economically consistent with the rest of the bank's forecasts, and generate the spreads between rates.

IAL's forecasting routines were used to generate the rates, the regression routines were used to check the relationships between rates and the I/O routines were used for storing these rates and spreads so that they could be accessed by the model.

Link into the bank's commercial loan system

Although our model was to be run on an outside time-sharing system, we needed large quantities of data from the bank's in-house commercial loan system. This system had been written in COBOL five years earlier and would have required many hours of modification before we could get our output directly from the operational system. Rather than spending an extensive amount of time on this problem, our solution was to have a program written in PL/1 to read the commercial loan tapes and write the necessary records in a format which could be easily read by FORTRAN. Some of the IAL modules were installed on our in-house batch system and then a series of IAL programs were written which simulated the loan portfolio for the next sixteen quarters, generated the proper schedules and rates for the outstanding loans and divided the loans into categories needed for the model. The data was then stored on the time-sharing system using the IAL I/O functions.

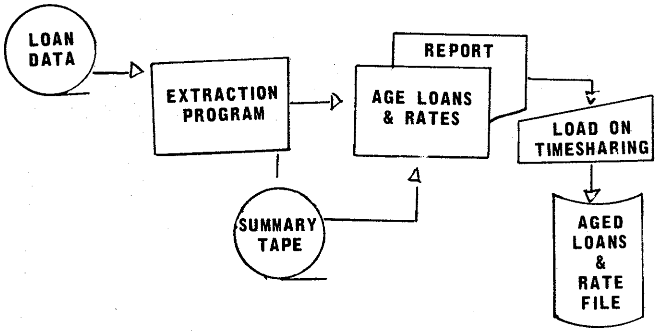


Figure 2—Link into Bank's commercial loan system

Gather direct expenses and income data

In order to generate salary expense tables, coefficients were needed to apply to hypothetical salary policies. The line of attack was to use a forecasting model on another time-sharing system to produce future levels for various loan and deposit types in the New York market. This was then broken down to bank and then departmental levels. Prior to this, historical loan and deposit information was gathered plus 150 time series of various activities performed within the banking operations groups. These series were the various activities that had to be performed to service the loan and deposit accounts. Using regression commands it was possible to determine the relationship between activity and account levels. The relationship between people and activity was also revealed. By starting at desired loan and deposit levels, then calculating the numbers of people needed and then finally using policy data, we could derive salary expenses.

Build the corporate planning model

The focal point of this effort is the Corporate Planning model. In order to design the model and to write the actual programs, an analyst who understood organizational structures and the accounting conventions of the bank was an absolute must. This was not a job for a programming or forecasting type. The task involved using all of the information provided by the other members of the Task Force and generating an earnings estimate and a tax strategy for the bank. In fact, this project was worked on in parallel with all of the other projects for the system. Because all of the input data had been stored using IAL, there was no problem of compatibility. Since record layouts are standardized as a result of using IAL, given the file name, the user is able to access any data on that file. Since all of the forecasting has been done in previous programs IAL was needed only for data manipulation, retrieval of tables stored on files, and printing of data.

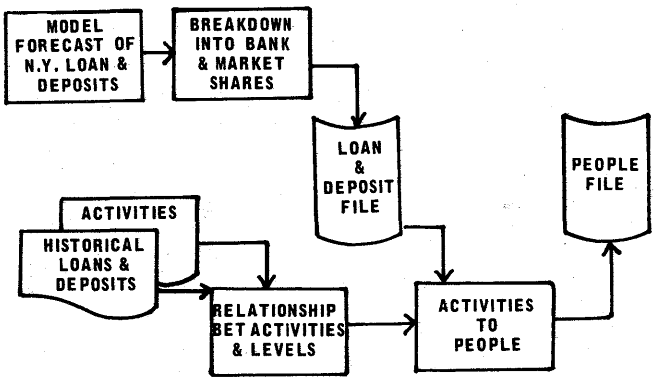

MODEL FORECAST OF N.Y. LOAN & DEPOSITS
BREAKDOWN INTO BANK & MARKET SHARES
LOAN & DEPOSIT FILE
PEOPLE FILE
ACTIVITIES
HISTORICAL LOANS & DEPOSITS
RELATIONSHIP BET ACTIVITIES & LEVELS
ACTIVITIES TO PEOPLE

Figure 3—Gather direct expense

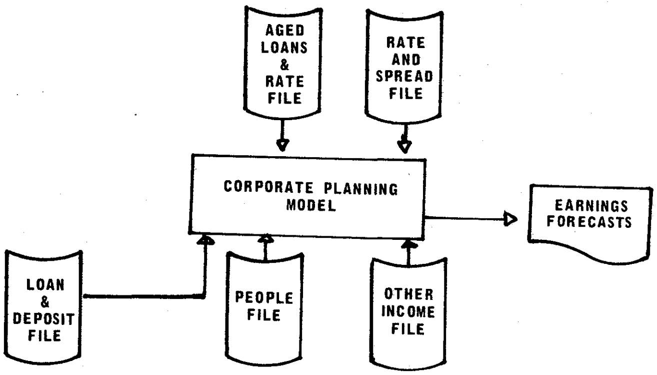


Figure 4—Build the corporate planning model

Generate the operating goals on a department level

The final output from the system was a set of operating goals for the various divisions within the bank.

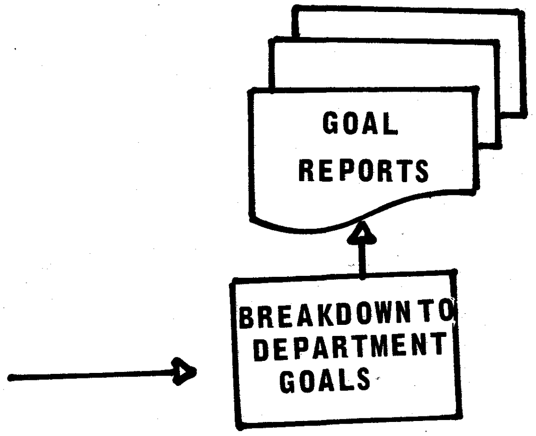


Figure 5—Generate departmental goals

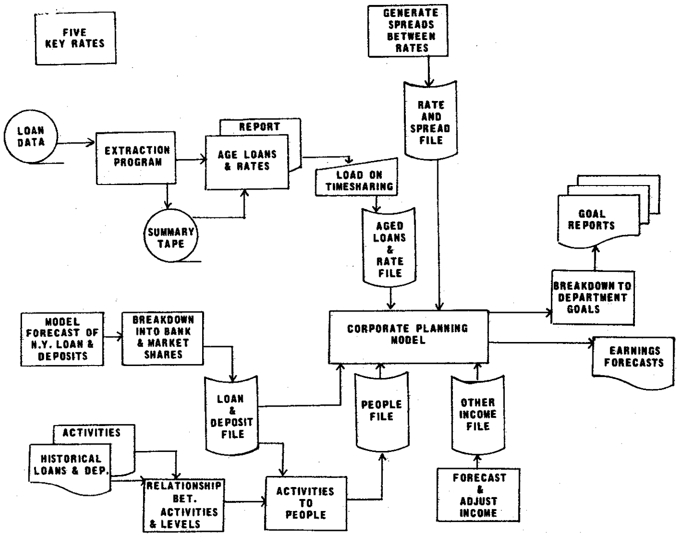


Figure 6—Corporate planning project

This program was assigned to a professional programmer because the expertise required for this project was a knowledge of data and file structures.

Review of the project

The programming and forecasting portions of this project were completed in two months time. A project of this magnitude could never have been completed in this period without the use of a language such as IAL. IAL allowed us to use each analyst where his expertise would be an asset to the project and programming *per se* was never an issue.

A project of this size usually is slowed down because team members have to wait while others complete their portions of the project. Because of the modularity of the major project design and the ease with which the user could store and retrieve data, data files and programs could be tested in parallel. This gave each analyst a chance to fine-tune his project and not have to rely on somebody else in order to meet his deadline.

The most difficult portion of the project was not communications within the team but the collection of data and learning the exact meaning of the collected data. Relieved of programming details, the group was able to devote more time to the real analysis side of the project.

BENEFITS GAINED FROM USING IAL

It is impossible to put a dollar value on the savings that the Management Science group has realized by using IAL for the last three years. This is primarily because the type of projects assigned to the department has changed radically. Prior to 1968 the group worked on short, one-time studies in various areas in the bank. After 1968 it was decided that the Management Science group should decrease the programming staff and have analysts work on a few key projects. The analysts were to do their own programming using IAL.

Less programming time

Although the projects assigned to the Management Science group were much more comprehensive after this change in policy, the programs used for analysis tended to be much shorter. A program composed of a series of calls to the IAL functions was usually less than a page (66 statements) in length. These short programs required less time in the debugging stage. The IAL functions had already been tested and certified so the user only had to make certain that the arguments used in the calls to the functions agreed with those of the language. Most errors occurred because the user did not read his manual carefully or because he made a mistake entering his parameters to the functions.

Cut time-sharing costs

The department's time-sharing expenses had been steadily increasing until the installation of IAL. Our expenses were cut by ⅓ after the advent of IAL. This included the decrease in terminal rental as well as a decrease in the cost for using commercial time-sharing systems.

Decreased need for documentation

Every analyst was forced to use IAL and as a result, to become familiar with the language. He was able to read any program written in the language. This benefited the group in two ways. There was little need for extensive documentation of a program and the only documentation required was a list of data to be provided by the user and a list of the results. With this, any analyst could pick up a project or work on the programs without extensive orientation.

As a result, the analyst had more time to spend on the structure of a problem. He did not have to worry about communicating his vague suspicions on how the program should be structured. He was on his own and in most cases loved it.

Aids in communication

One of the biggest advantages to be gained from using IAL is that everybody is speaking the same language. IAL is actually enforcing a set of definitional and computational standards on the user group. As an example, all users will be calculating depreciation schedules, internal rates of return and present values using the same methods. The benefits gained from a common language can be spread throughout the organization as well as within an individual group. At Bankers Trust the Management Science Division, Economics Division, BT Consultants (financial consultants) and the Credit Analysis Group all use IAL. The groups are able to share programs without extensive documentation, discuss the solutions to problems and in some cases, implement the solutions without overwhelming communications problems. In this way any organization can leverage the unique talents of various groups within it and develop company-wide projects.

Today there are many good packages and languages available for various types of analysis. Packages have been designed to run regressions, to perform statistical analysis and to calculate present values. These packages are good but IAL has the capability of operating in all of these areas. This means that every professional in a research group, regardless of his expertise, is speaking the same language. This is one of the strongest arguments for using IAL.

PROBLEMS WITH THE LANGUAGE

Terminal I/O

The obvious drawbacks are those that meet the user's eye first. Until recently, the terminal I/O commands were very brief and there was no generalized report-generating capability. The language was designed as a tool for research and as such did not need options for elaborate reports.

Another drawback was that the user was initially providing input on-line as his program was running. In many cases, depending on the time-sharing system when the user accidentally typed an alphabetic character for a number he would be forced to start over and rerun the program.

These two problems have been solved first by adding a generalized report generator to the list of commands and secondly designing a function which allows the user to enter data through the editor of the time-sharing system instead of on-line.

Extended lists

Because of the need to pass tables of various lengths and dimensions it was necessary to design the system around extended lists instead of tables. An extended list is devised for a table of M by N dimensioning and storing it in a list of M*N length. Element (1, 2) of the table is stored in element of M+1 of the list. This presents no problems to those using all IAL functions in a program. If a user wishes to mix his own code with IAL he must be very careful when moving elements in the extended lists to make sure that they would correspond to the correct element in his imaginary table.

New users

If a potential user of IAL has learned to use BASIC or FORTRAN he usually feels that he is a pretty good programmer and is generally more of a problem than somebody who has never programmed. A person with previous experience feels he doesn't need help and the idea of using a package is pretty silly. The only way that this can be solved is to have him working with somebody using the language. He will observe how easy it becomes to use the functions and gain confidence in the language.

After a while he will get a great deal of satisfaction out of being able to do the programming end of the project with such ease.

Distribution of IAL

IAL has been leased to The American Bankers Association by Mr. Ahlers. The ABA is distributing the system to commercial time-sharing vendors throughout the United States and hopefully IAL will be marketed world-wide. IAL has been installed on seven time-sharing systems and is available to any vendor who wishes to install the language, provided he goes through the prescribed certification process. This process was instituted to guarantee the integrity of the computational routines after installation on a new system. In most cases IAL is available at no extra charge to any subscriber on any one of the seven time-sharing systems. Plans are now being processed to make IAL available to the universities and it will soon be used in the graduate schools at Carnegie-Mellon University and Harvard University.

After IAL becomes available as a teaching tool in the business schools, more and more young anaylsts will find it an efficient means of performing most of their financial analysis.

The American Bankers Association has also offered week-long courses in the use of IAL and the financial principles behind the language. These courses have been attended by many members of the Management Science and Operations Research groups from all of the major banks in the United States.

A users manual for the language has also been written and is distributed through the ABA.

FUTURE OF IAL

The IAL system has been a success in the financial and research areas. A demand has been developed for new functions which will fit into the IAL framework but deal with very specialized areas of banking or mathematics. As described earlier, various modules have been added: corporate tax modules, bond modules and generalized report generators. Functions are being added to the system to solve linear programming problems and for use in adaptive forecasting. IAL was originally designed to be used for short research problems but I feel that its real future lies in serving as a communications link in large projects. This can be the unifying thread in a large project such as the corporate planning model designed at Bankers Trust. The language has proven to be an ideal means of communications within a group or corporation.

ACKNOWLEDGMENT

I would like to thank Dave Ahlers for the opportunity of working on such an interesting project. I am also indebted to Judith Liersch and Arlene Kaplan for their tireless efforts in re-reading, and re-reading, and re-reading this paper.

REFERENCES

1 D M AHLERS
IAL reference manual
American Bankers Association 1970

2 C A DMYTRYSHAK
The development of the investment analysis language
MS Thesis Department of Computer Science Pratt Institute June 1970

The design approach to integrated telephone information in the Netherlands

by R. A. DiPALMA*

Litton—Mellonics
Sunnyvale, California

and

G. F. HICE

PANDATA N.V.
Rijswijk (ZH), the Netherlands

INTRODUCTION

The Integrated Telephone Customer Information System (ITCIS) is a computer network system, which was initiated by the Dutch Post, Telephone and Telegraph (PTT) and PANDATA N.V., a Dutch software company partly owned by PTT, in June 1970. The initial definition study concerned the feasibility of integrating several data files each containing telephone customer data (Billing, Directory Preparation, and Work Order Administration). In addition, there were efforts under way by a PTT research group concerning the automation of the Directory Assistance Service and, by another group, the Telephone Cable and Pair Administration. The conclusion reached in the definition study indicated that integration was not only feasible, but, that a completely integrated on-line system, including Cable and Pair and Directory Assistance would be economically desirable.

Subsequently, the Preliminary Design of ITCIS was undertaken in October 1970. Within this design stage, hardware and software elements were designed for a system based on the projected workload for 1980 when, it was estimated, there would be more than 4 million telephone customers. The applications included within the integrated system are:

- Directory Assistance Inquiry
- Directory Preparation
- Billing and Collections
- Work Order Entry and Administration
- Customer Services Inquiry
- Cable and Pair Administration and Inventory
- Management Information.

These applications involve large batch runs coupled with high load real-time inquiry.

During the Preliminary Design effort, a number of trial configuration approaches were developed and examined to determine the most favorable approach for ITCIS. A decision matrix analysis, combined with a method for converging opinions, was used. The result of this analysis is that a configuration approach based on a centralized data base, on-line to all administrative districts is the most advantageous. The hardware facility required to support this centralized data base, designated the **C**entral **P**rocessing **F**acility (CPF), is a multi-processor mainframe using large capacity removable disk storage to contain the data base.

Access to this shared centralized facility will be provided via dedicated communications circuits between each district and the CPF. These circuits will be terminated in each district by a small general purpose computer, designated a **C**omputer **B**ased **C**oncentrator (CBC) which will act as data concentrators and remote batch terminals.

There are over 150 points at which a remote device must be located. These include 13 telephone district

* Formerly of PANDATA N.V.

headquarters, 20 directory assistance operator rooms and 120 technical service areas. Each district headquarters requires remote batch facility and a minimum of one teletypewriter (TTY) and visual display unit (VDU). A cluster of up to 40 VDU's is needed in each of 20 operator rooms. Each of the 120 service areas requires a minimum of one TTY.

Some generalized system software, other than that supplied by the manufacturer, will be required to support ITCIS application programs, in the areas of:

- District computer operating system and communications
- Real-time communications and processing at the CPF

Detailed design and programming of the first phase of implementation of ITCIS has now begun. This phase will automate Directory Assistance (Inquiry 008) and Cable and Pair.

There are several points of interest in this design effort which will be discussed in more detail. These include the hardware/software environment mentioned briefly above, some of the design techniques used, and the use of a manufacturer-implemented version of the CODASYL Data Base Task Group Report.[1]

HARDWARE/SOFTWARE ENVIRONMENT

The CPF proposed for the final system is a UNIVAC 1110 multi-processor with three subsystems of 8440 disk storage (2.5 billion characters). Two Control units are used with each disk subsystem. It is a 2×2 system with 2 Command/Arithmetic Units (CAU) and 2 I/O Access Units (IOAU). Memory consists of 128K words of Main Memory (plated wire) and 262K words of Extended Core Memory. The communications interface consists of 2 Communications Terminal Module Controllers (CTMC) and 8 Communications Terminal Module (CTM). A tape subsystem of 6 drives is included. Unit record I/O is handled by 2 UNIVAC 9300's with printers and card reader/punches. See Figure 1 for the CPF configuration.

The remote processors (CBC) will most likely be PDP 11/20's. Each PDP 11 (15 in all) will act as remote batch terminal, accepting card and paper tape input and producing printed output. The main function, however, consists of being an on-line communications multiplexor and concentrator. The configuration consists of 8K (16 bit) words of memory augmented by 64K words of disk storage. Card, Print and paper tape I/O is provided, as well as interfaces for the required terminals and high speed link to the CPF. See Figure 2 for the CBC configuration.

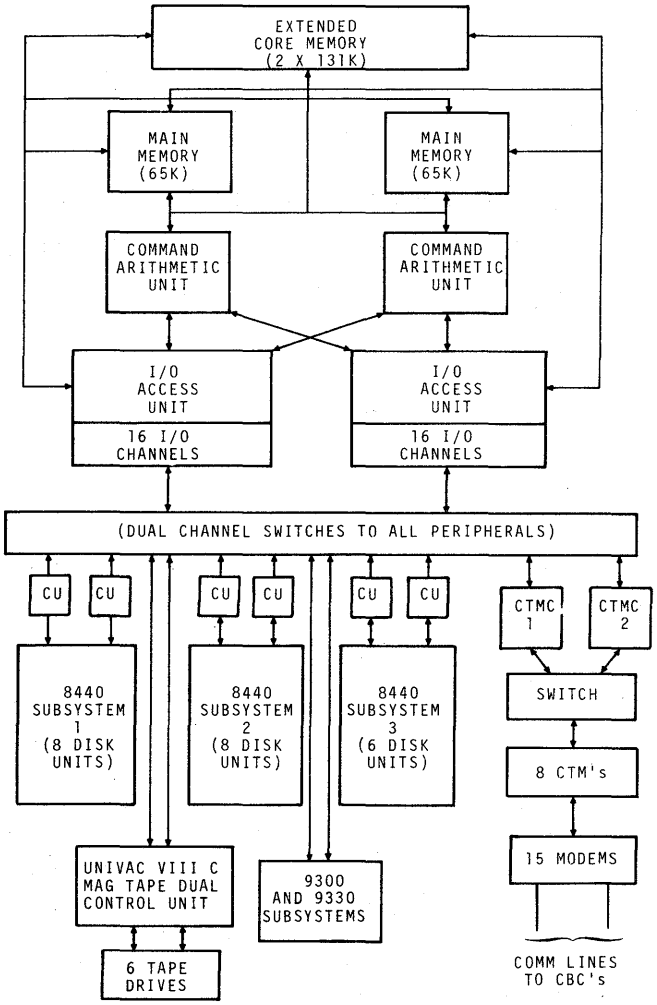


Figure 1—CPF configuration

The high speed link will be leased lines (15) operating at 2400 and 4800 bps, full duplex. The communications technique used is segmented messages with acknowledgement of first, last and retransmitted segments. Cyclic error protection codes are used.

There are over 600 terminals required (in 1980). Of these, 425 are Uniscope 100's for Directory Assistance Inquiry and Customer Service Inquiry. Both multistation and stand-alone connections are allowed with remote connections via 2400 bps lines and modems and local connections via 4800 bps cables. There are almost 200 Automatic Send/Receive (ASR) teletypewriters situated in customer service areas and connection departments for work order entry, maintenance and

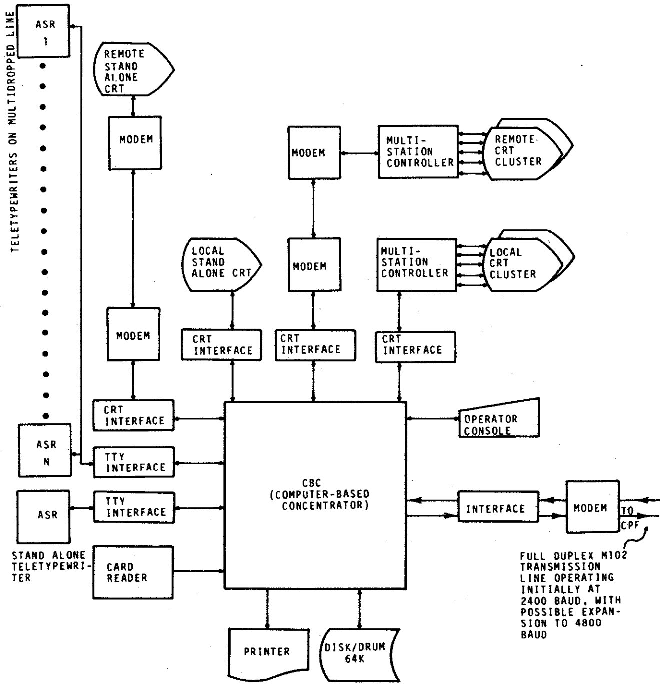


Figure 2—CBC configuration

inquiry. The TTY's are connected to the CBC's over local leased 110 bps lines and may be connected in multi-drop or stand-alone fashion. Figure 3 shows the approximate location of terminals throughout the Netherlands.

The total configuration is to be built up in a modular way over the period of 1972 to 1978 according to application implementation and system growth. The stress has been on flexibility, modularity and reliability.

The emphasis in design of the software for ITCIS has also been on flexibility, modularity and reliability. The software to implement ITCIS has either been selected from available general purpose UNIVAC software, or is being developed as a joint effort of PANDATA and PTT personnel.

In order to meet the requirements cited above, as much general purpose UNIVAC software as possible is being used. To qualify, the software must meet performance requirements and be available now or in the immediate future. The software being used from UNIVAC is:

- *EXEC-8* The operating system in its entirety and without modification, is being used to control the CPF resources. This means that ITCIS is not a dedicated system but can be shared by other applications, perhaps even other real-time applications.
- *DMS-1100* Is also being used in its entirety. The run time component of DMS-1100, called DMR, is presently available as a single thread program, with a reentrant version, capable of concurrent run-unit execution, to be available in June 1972. Figure 4 shows how the components of DMS-1100 are used to build a SCHEMA and an application program to manipulate the Data Base.
- *RTS* UNIVAC Real Time Scope Handler, has been modified by PANDATA to activate the user whenever a message comes in on any communication line, to dynamically handle message buffers and to provide a general pool capability.

Figure 5 shows how both jobs and inquiries are initiated in this system. Figure 6 shows the interface to the data base.

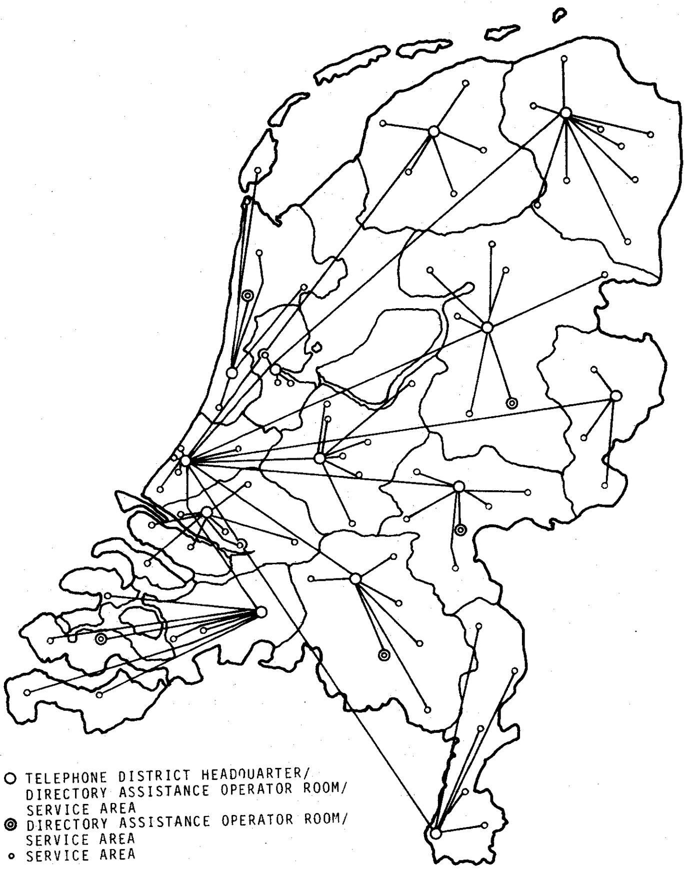
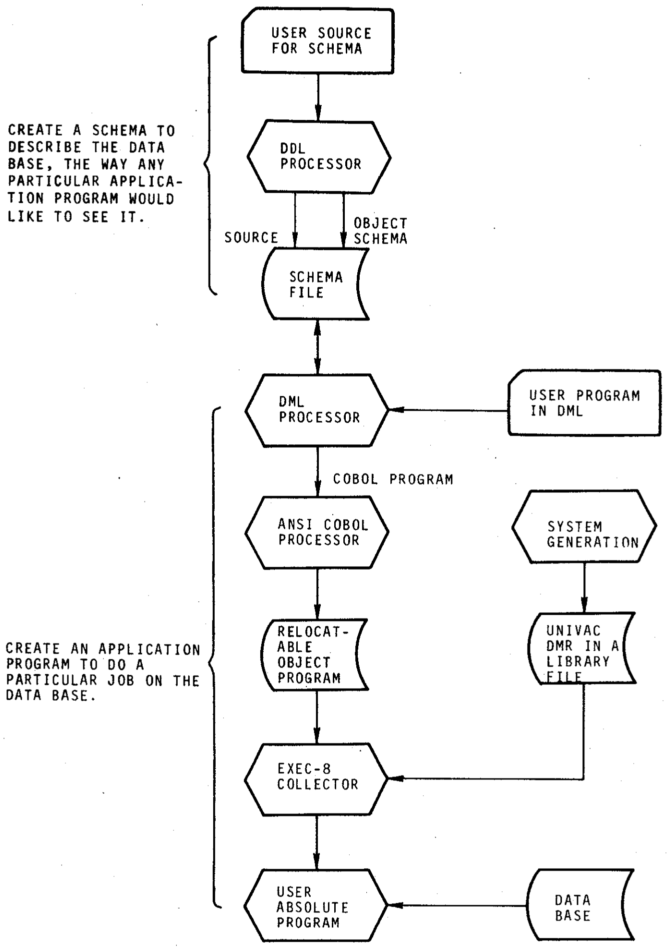

○ TELEPHONE DISTRICT HEADQUARTER/ DIRECTORY ASSISTANCE OPERATOR ROOM/ SERVICE AREA
◎ DIRECTORY ASSISTANCE OPERATOR ROOM/ SERVICE AREA
∘ SERVICE AREA

Figure 3—Telephone districts of the Netherlands, showing network

The application software being developed for ITCIS is also indicated and explained in Figures 5 and 6. It should be noted that the scheme is open-ended enough so that new applications can be added to the ITCIS network by simply connecting new terminals to the CBC's and adding real-time transaction processors or batch runs needed to the CPF software.

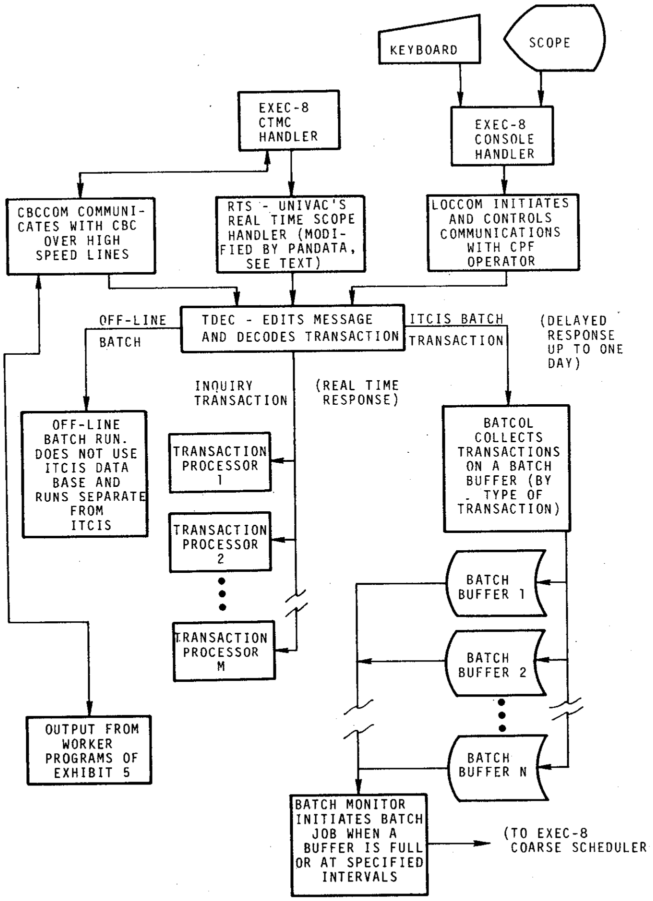
DESIGN TOOLS

The entire design effort has been conducted within the framework of the PANDATA *System Development Methodology* (SDM) which is a standard controlling the design and development of large-scale systems. This method provides detailed sets of activities within each stage of seven stages of system development. For each activity, there are explicit steps that must be performed which result in specific products and require specific inputs. When followed closely, the SDM has provided high quality and complete documentation, and has resulted in well controlled timely development efforts.



USER SOURCE FOR SCHEMA
CREATE A SCHEMA TO DESCRIBE THE DATA BASE, THE WAY ANY PARTICULAR APPLICATION PROGRAM WOULD LIKE TO SEE IT.
DDL PROCESSOR
SOURCE
OBJECT SCHEMA
SCHEMA FILE
DML PROCESSOR
USER PROGRAM IN DML
COBOL PROGRAM
ANSI COBOL PROCESSOR
SYSTEM GENERATION
CREATE AN APPLICATION PROGRAM TO DO A PARTICULAR JOB ON THE DATA BASE.
RELOCATABLE OBJECT PROGRAM
UNIVAC DMR IN A LIBRARY FILE
EXEC-8 COLLECTOR
USER ABSOLUTE PROGRAM
DATA BASE

Figure 4—Use of DMS-1100 to create a data base SCHEMA and an application program



KEYBOARD
SCOPE
EXEC-8 CTMC HANDLER
EXEC-8 CONSOLE HANDLER
CBCCOM COMMUNICATES WITH CBC OVER HIGH SPEED LINES
RTS - UNIVAC'S REAL TIME SCOPE HANDLER (MODIFIED BY PANDATA, SEE TEXT)
LOCCOM INITIATES AND CONTROLS COMMUNICATIONS WITH CPF OPERATOR
OFF-LINE BATCH
TDEC - EDITS MESSAGE AND DECODES TRANSACTION
ITCIS BATCH TRANSACTION
(DELAYED RESPONSE UP TO ONE DAY)
INQUIRY TRANSACTION
(REAL TIME RESPONSE)
OFF-LINE BATCH RUN. DOES NOT USE ITCIS DATA BASE AND RUNS SEPARATE FROM ITCIS
TRANSACTION PROCESSOR 1
TRANSACTION PROCESSOR 2
TRANSACTION PROCESSOR M
BATCOL COLLECTS TRANSACTIONS ON A BATCH BUFFER (BY TYPE OF TRANSACTION)
BATCH BUFFER 1
BATCH BUFFER 2
BATCH BUFFER N
OUTPUT FROM WORKER PROGRAMS OF EXHIBIT 5
BATCH MONITOR INITIATES BATCH JOB WHEN A BUFFER IS FULL OR AT SPECIFIED INTERVALS
(TO EXEC-8 COARSE SCHEDULER

Figure 5—Software environment—CPF inquiry and batch job initiation

In addition to the general design philosophy supported by SDM for ITCIS, two Operations Research tools were used during Preliminary Design in a rather unorthodox, but effective, and therefore interesting manner. The tools themselves were not unusual at all to problems in System Design or business—Decision Theory and Simulation.

USE OF DECISION THEORY

Initially, there were a wide variety of basic system approaches (both centralized and decentralized) which could have been used for ITCIS. In order to resolve such controversial and subject problems, a "payoff"

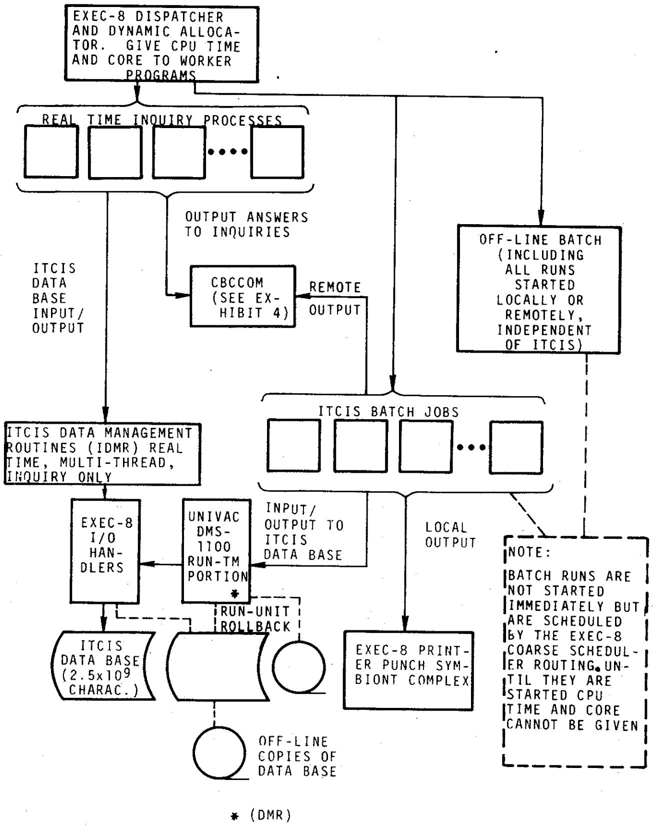

EXEC-8 DISPATCHER AND DYNAMIC ALLOCATOR. GIVE CPU TIME AND CORE TO WORKER PROGRAMS
REAL TIME INQUIRY PROCESSES
OUTPUT ANSWERS TO INQUIRIES
ITCIS DATA BASE INPUT/ OUTPUT
CBCCOM (SEE EXHIBIT 4)
REMOTE OUTPUT
OFF-LINE BATCH (INCLUDING ALL RUNS STARTED LOCALLY OR REMOTELY, INDEPENDENT OF ITCIS)
ITCIS BATCH JOBS
ITCIS DATA MANAGEMENT ROUTINES (IDMR) REAL TIME, MULTI-THREAD, INQUIRY ONLY
EXEC-8 I/O HANDLERS
UNIVAC DMS-1100 RUN-TM PORTION *
INPUT/ OUTPUT TO ITCIS DATA BASE
LOCAL OUTPUT
NOTE: BATCH RUNS ARE NOT STARTED IMMEDIATELY BUT ARE SCHEDULED BY THE EXEC-8 COARSE SCHEDULER ROUTING. UNTIL THEY ARE STARTED CPU TIME AND CORE CANNOT BE GIVEN
RUN-UNIT ROLLBACK
ITCIS DATA BASE (2.5x10⁹ CHARAC.)
EXEC-8 PRINTER PUNCH SYMBIONT COMPLEX
OFF-LINE COPIES OF DATA BASE
* (DMR)

Figure 6—Software environment—CPF inquiry and batch job processing

matrix was used in the following manner:

1. Four alternative configurations were designed, varying from completely decentralized (thirteen stand-alone processing facilities, one for each telephone district in the Netherlands) to completely centralized (all computing done at a central facility). These four configurations define rows of the decision matrix (see Figure 7).
2. A number of detailed Selection Criteria were established in the categories shown on Figure 7.
3. Value Analysis followed. Members of the ITCIS Preliminary Design Project Staff and other interested parties were asked to rate the criteria according to value.
4. Efficiency analysis was conducted with a small group more familiar with the trial configurations; efficiencies of each approach toward meeting each criteria were established and quantified. In some cases (e.g., cost) a quantification was simple and direct—in others quantification was more subjective but could sometime be related to cost. For example, the reliability of two approaches can be made equal by spending more money on one of the approaches.
5. The coordinator of the analysis (one of the authors) carried arguments to various participants. As a result the deviation of many responses was lessened.
6. The standard deviation high, low and mean figures for efficiency and value were put into the efficiency matrix and the value vector, and 3 matrix multiplications were performed, using means, high and low figures.
7. The results were clearly in favor of the two centralized approaches, although the unit processor and the multi-processor options were hard to distinguish. A choice was later made for the multi-processor because in the configuration required it is cheaper, and the modular architecture of the UNIVAC 1110 (UNIVAC was the most serious contender because PTT had

DECISION MATRIX

SELECTION CRITERIA → CONFIG-URATION APPROACH ↓	CRITERION 1	CRITERION 2	. . .	CRITERION i	. . .
APPROACH A 13 COMPUTERS, CONNECTED TO SERVICE INTER-DISTRICT TELEPHONE NUMBER INQUIRIES			. . .		. . .
APPROACH B SEVERAL COMPUTERS (BUT NOT 13) INTERCONNECTED AND SHARING A CENTRAL DATA BASE, BUT EACH CAPABLE OF STAND-ALONE OPERATION			. . .	EFFICIENCY OF APPROACH B TOWARD MEETING SELECTION CRITERION i	. . .
APPROACH C MULTIPLE UNIT PROCESSORS AT A CENTRAL FACILITY WITH A REDUNDANT SHARED DATA BASE			. . .		. . .
APPROACH D A SINGLE MULTI-PROCESSOR SYSTEM (AS DESCRIBED IN PART II)			. . .		. . .

SELECTION CRITERIA CATEGORIES:

COST
RELIABILITY
SECURITY
DESIGN FACTORS
ERGONOMIC FACTORS

Figure 7—Decision matrix

a UNIVAC 1106 already, which was to be used for development) is almost as internally redundant as completely separate computers.

The Decision Theory Analysis, although still subjective in some cases, reduced subjective decision to such a low level that most participants were unable to be swayed by prejudice toward the much more encompassing (but still underlying) question of centralization vs. decentralization.

USE OF SIMULATION

Design validation for a system as large and complex as ITCIS is very difficult and simulation of ITCIS was one of the primary methods used to answer design questions. Very early in Preliminary Design a GPSS-II model (GPSS-1100 was not yet available) was written of the ITCIS system, to investigate CPF throughput, line utilization, etc. It was soon found that the ITCIS model was pushing the standard version of GPSS-II to its limit. GPSS-II extended version would not run on the PTT 1106 because the core was too small and, in addition, GPSS was not suitable for such a highly interactive, complex system. Certain useful results were still obtained but more simulations were planned for detailed design to examine the behavior of the CPF under overloads, etc., as the hardware and software design become firmer.

Splitting the model into submodels was considered but the entire ITCIS is very interactive due to the communications technique and the nature of Inquiry 008, i.e., multiple operator/computer interactions due to a single telephone number inquiry.

Of the available UNIVAC simulation packages, systems and languages, SIMULA I (which was developed at the Norwegian Computing Center) was selected because:

- It was supported as standard UNIVAC software (unlike SIMSCRIPT and GASP)
- It allowed similar model structure to GPSS (i.e., *flow* or *process* oriented)

In fact the SIMULA language is so powerful that a model can be organized in a manner very much like the required organization of other modelling systems. (SIMULA 67 was not yet available but would have been even more suitable.)

As a result a special purpose simulation system was developed. (See Figure 8.) It is based in SIMULA, to be used to create models of ITCIS by changing the parameters of an input data set. (This can be done easily by using a system utility.) This simulation system was developed so that changes to ITCIS could easily be incorporated in the model without reprogramming, and a great deal of detail could be inserted in certain areas of the model without affecting other areas.

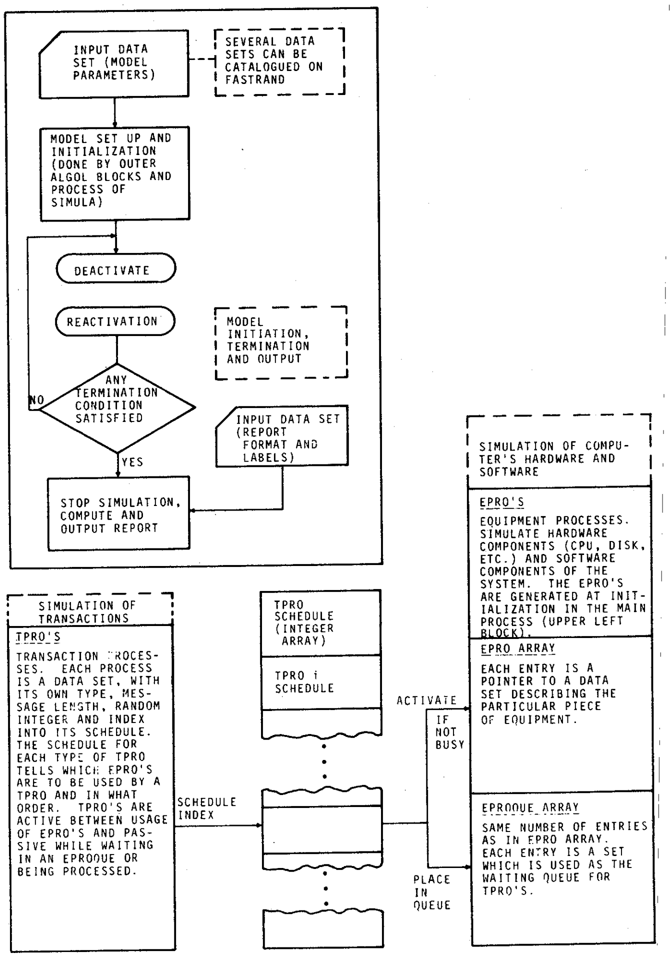

INPUT DATA SET (MODEL PARAMETERS)
SEVERAL DATA SETS CAN BE CATALOGUED ON FASTRAND
MODEL SET UP AND INITIALIZATION (DONE BY OUTER ALGOL BLOCKS AND PROCESS OF SIMULA)
DEACTIVATE
REACTIVATION
MODEL INITIATION, TERMINATION AND OUTPUT
ANY TERMINATION CONDITION SATISFIED
NO
YES
INPUT DATA SET (REPORT FORMAT AND LABELS)
STOP SIMULATION, COMPUTE AND OUTPUT REPORT
SIMULATION OF COMPUTER'S HARDWARE AND SOFTWARE
EPRO'S
EQUIPMENT PROCESSES. SIMULATE HARDWARE COMPONENTS (CPU, DISK, ETC.) AND SOFTWARE COMPONENTS OF THE SYSTEM. THE EPRO'S ARE GENERATED AT INITIALIZATION IN THE MAIN PROCESS (UPPER LEFT BLOCK).
SIMULATION OF TRANSACTIONS
TPRO'S
TRANSACTION PROCESSES. EACH PROCESS IS A DATA SET, WITH ITS OWN TYPE, MESSAGE LENGTH, RANDOM INTEGER AND INDEX INTO ITS SCHEDULE. THE SCHEDULE FOR EACH TYPE OF TPRO TELLS WHICH EPRO'S ARE TO BE USED BY A TPRO AND IN WHAT ORDER. TPRO'S ARE ACTIVE BETWEEN USAGE OF EPRO'S AND PASSIVE WHILE WAITING IN AN EPROQUE OR BEING PROCESSED.
TPRO SCHEDULE (INTEGER ARRAY)
TPRO i SCHEDULE
SCHEDULE INDEX
ACTIVATE
IF NOT BUSY
PLACE IN QUEUE
EPRO ARRAY
EACH ENTRY IS A POINTER TO A DATA SET DESCRIBING THE PARTICULAR PIECE OF EQUIPMENT.
EPROQUE ARRAY
SAME NUMBER OF ENTRIES AS IN EPRO ARRAY. EACH ENTRY IS A SET WHICH IS USED AS THE WAITING QUEUE FOR TPRO'S.

Figure 8—Major components of ITCIS simula modeling system

This much generality was felt necessary for the ITCIS project, but the resulting system is in fact useful for modelling any teleprocessing system, and there are plans to use it in other systems.

The only model completed to date was a model very similar to the GPSS-II model. The results are compatible although more extreme situations show up in the SIMULA model. The SIMULA program takes about one-third of the core required by the GPSS model and executes in about 85 percent of the time required by the GPSS program.

SUMMARY

The ITCIS System, besides the usual expected advantages of providing better customer service and maintaining lower operating costs, has resulted in the testing of a System Development Methodology, the development of a Simulation tool which will make simulating computer systems several orders of magnitude easier, and the synthesization of a decision theory technique which should prove useful in many subjective situations. In addition the value of using Economies of Scale has been demonstrated, in that a large system, designed to handle a peak inquiry load which will only last a few hours a day, will be used during non-peak hours to handle all other customer services.

By the end of June 1972, a SCHEMA had been written using the DDL of DMS-1100, and trial runs indicated that use of this large, general purpose system is feasible. The possible availability of a COBOL compiler which can generate re-entrant, asynchronous program code also indicates that it may be possible to use COBOL to a much greater extent than normally thought possible in systems programming.

REFERENCE

1 October 1969 Report of the CODASYL Data Base Task Group of the CODASYL Programming Language Committee

Field evaluation of real-time capability of a large electronic switching system

by W. C. JONES and S. H. TSIANG

Bell Telephone Laboratories
Naperville, Illinois

INTRODUCTION

The Bell System's No. 1 Electronic Switching System[1] (No. 1 ESS) was designed for medium-to-large telephone offices. Its performance has been improved radically since first put into service on May 30, 1965, in Succasunna, New Jersey. By June, 1972, some 250 No. 1 ESS offices were in service equipped with over 4 million customer lines.

This paper describes a load test which was conducted recently in a field office to evaluate the real-time capability of the latest program, named SPCTX-5.

In order to aid the reader in the comprehension of this paper, some background information is provided.

BACKGROUND

General

Capacity of a telephone system is multidimensioned. It can be measured in terms of quantity of calls processed, number of customers served, traffic load handled by the switching network, etc. The No. 1 ESS capacity, to date, has been limited only by the capability of the processor and its associated program. The number of calls that the system can handle is directly proportional to the amount of time that it takes to process individual calls.

History on call capacity

In the past 6 years, a great deal of effort has been expended to increase the call-carrying capacity of the No. 1 ESS as well as to add new features. Improvements were made through both hardware and software means. Of the several developments that have produced a substantial increase in the traffic-handling capacity of No. 1 ESS, the streamlining of the program is perhaps the most significant.

Figure 1 shows a history of the No. 1 ESS call capacity improvements. In 1965 and 1966, we had only two No. 1 ESS offices in service, and the maximum call carrying capacity at the time was about 25,000 peak busy hour calls. This is only an estimate since few meaningful measurements were taken. Peak busy hour calls are the number of calls estimated for an office on its highest normally recurring busy hour during the busy season. This number must be known in order to engineer the office properly. It is generally assumed that of the number of peak busy hour calls, about 85 percent complete to talking and about 15 percent to busy or no answer.

The capacity of the system is usually expressed in terms of a range—maximum, average, and minimum. This is because the type of traffic handled varies from office to office. For example, an office may have more interoffice calls than intraoffice calls. The machine time consumed by processing different types of calls are different.

The program released at the beginning of 1967, has a capacity of about 27,000 peak busy hour calls. Soon after, program improvements were made to increase the call capacity to 32,000.

In 1968, a significant increase in capacity was achieved through the addition of a signal processing unit (SP) to the basic central control processing unit (CC), to take over many of the repetitive functions such as scanning lines and collecting dialed digits. SP is essentially an input/output processor. With SP, the system reached a capacity of about 64,000. In 1969, further program improvements were made which increased the capacity to 71,000.

As more experience was gained, it became clear that the central control processor was spending most of its

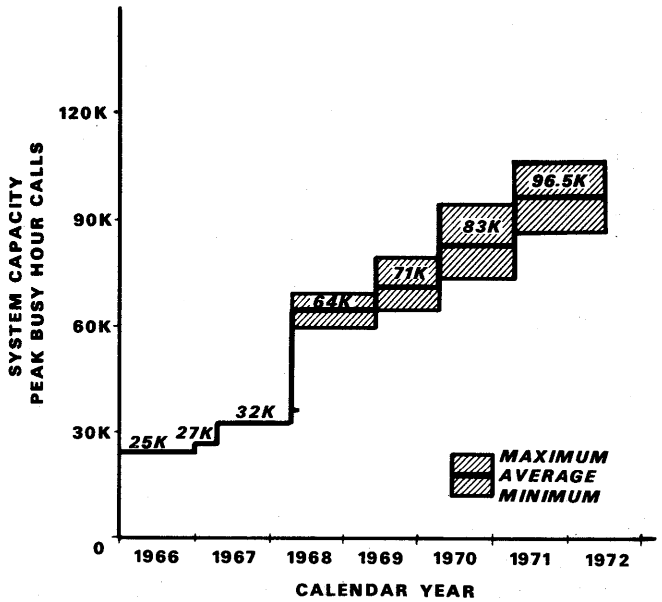


Figure 1—No. 1 ESS capacity

real time in network connections. To simplify some of these actions, the service link network was developed. This equipment is a new adjunct to the standard switching network, and is designed to simplify the operations that set up ringing and digit-receiver connections. The service link network hardware relieves the No. 1 ESS program of much of the chore of establishing these routine connections and, thus, increases the call capacity by greatly reducing the average time the program is spent with each call. Introduction of the service link network in 1970, raised the No. 1 ESS capacity to 83,000. Major program improvements were made in 1971, which resulted in a maximum capacity of over 100,000. This was the original No. 1 ESS design objective.

It is significant that these capacity improvements have been made at the same time that many new features, which tend to reduce capacity, were being added. The program introduced at the beginning of 1967, has 137,000 instructions (44 bits/instruction). The latest program, which has incorporated many new features as well as fault detection and diagnosis for the new hardware and call capacity improvements, has over 230,000 instructions of which over half are required for maintenance.

Capacity studies

Figure 2 shows a greatly simplified model of the real-time usage in No. 1 ESS. The number on the ordinate shows the real-time consumption; the bottom half represents 100 percent of CC real time, and the top half 100 percent of SP real time. The SP performs all the input/output (I/O) work, and the CC executes all other work associated with each call.

The abscissa shows peak busy hour calls. As shown in this Figure, different job segments comprise the total real time consumed. The SP overhead and the CC overhead are constant, relatively independent of the amount of traffic being processed. The equipment dependent I/O real-time consumption is directly related to the amount of equipment, such as lines and trunks, in the office. A trunk is a circuit which provides a communication channel between telephone offices. The line representing the equipment dependent I/O indicates that the real-time usage increases as calls per hour increase. The lines for the per call I/O and per call other work show the percent of real-time consumption also rising as the level of calls goes up. The slopes of these lines depend upon the average amount of real time consumed per call. As program improvements are made, the slopes decrease. As a result, the call capacity goes up.

A number of techniques have been developed to perform real-time studies. Simulation[2] is one of these techniques. A somewhat simpler method is to determine the number of machine cycles required for overhead and for processing various types of calls such as intraoffice

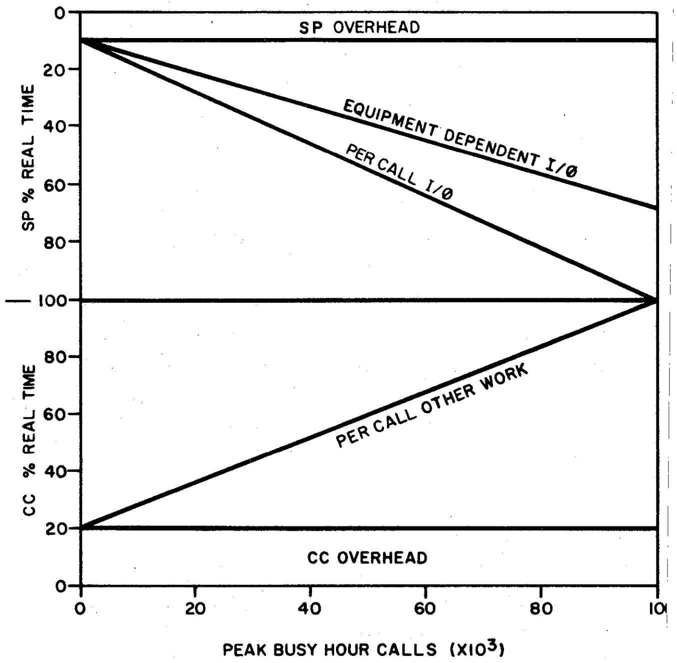


Figure 2—No. 1 ESS real-time usage

with TOUCH-TONE®, outgoing with multifrequency pulsing, etc. These cycle counts can then be used to estimate the call carrying capacity. In early days, both call-type cycle counts and capacity calculations required a large amount of manual effort. Since there are many different types of calls, both jobs were time consuming and tedious. Now, all have been automated.

In the automated procedure, a programmable electronic call simulator controlled by a computer system, is used to generate a set of test calls in the system laboratory. While a call is being processed, machine cycles are counted by a program-controlled counter. ESS utility programs collect and record the counts on a magnetic tape. This tape is then processed on a commercial computer which prints out the cycle counts.

A total of 50 call-type cycle counts have been collected. Since each type of call is processed by the system in stages separated by time breaks, the call-type cycle count is made up of cycle counts of many program segments. The segment cycle counts are summed to determine the real time required for each type of call.

The overhead and call-type cycle counts form the data base for a capacity estimating program ESS1CAP, and are used for computing the call processing capacity of No. 1 ESS offices. ESS1CAP is a conversational, time-shared computer program. Capacity is estimated in the manner similar to manual calculations employed previously, except that the manual effort is greatly reduced. In the manual method, the number of input items that the telephone company traffic engineer had to provide was great—well over a hundred. Now only about 20 input items are needed for the ESS1CAP program. These are the traffic mix of the office for which the peak hour call capacity is to be estimated. Traffic mix includes such items as percent originations of total calls, and a further breakdown of originations into partial dials, intraoffice, and outgoing calls. ESS1CAP is also an important tool for evaluating capacity improvements. Our experience with ESS1CAP capacity predictions has been good. However, its accuracy had not been fully verified under controlled load conditions.

LOAD TEST

Purposes

Although we were reasonably sure about our call capacity estimates, we felt that there is no substitute for a load test with real calls. Some of the reasons for the load test are to verify the real-time improvements and to check the accuracy of the cycle count data collected with the newly automated procedures. Such a test is also important to secure the confidence of the telephone operating companies in the use of the ESS1CAP program, which is made available to them for call capacity estimates of their No. 1 ESS offices. Another reason for performing the load test is to determine the adequacy of the present overload control with the increased call capacity.

In overload control, the executive control program must ensure that peaks of traffic do not overwhelm the system and cause it to process less than an optimal load. An overload control program can modify the operation of the executive control. Under overload conditions, scanning for new service requests is slowed down and the hopper which is used to store these service requests, is emptied much less frequently. It is both convenient and desirable to handle overload by limiting the traffic. Each additional service request is a commitment by the system to perform a certain amount of data processing. By an orderly deferral of further commitment during overload, we guarantee that the data processing overload is rapidly eliminated. Otherwise, the delays in processing calls becomes so great that many customers hang up before their calls have been completed. This wastes that portion of call processing which had already been completed, and leads to further overloads when these customers try again to complete their calls. The overload control has been simulated on a general-purpose computer; however, it has not been fully verified in the field under overload conditions.

Environment and equipment

The load test was conducted in an office at Portland, Oregon, prior to its cutover into service. The system was running very well at that time with no obvious call-effecting hardware or software problems. There was a sufficient number of trunks and service circuits available in that office to make the test possible. Service circuits include items such as signal transmitters, digit receivers, ringing and other similar circuits. A total of 1100 test lines and 1000 trunks were employed in the test.

The calls are generated by 11 load boxes. Load box is a type of test set (Figure 3). Each set can originate calls on 50 lines which are divided into ten groups of five lines each. A maximum of 13 digits may be pulsed over each group. All five lines in the same group will dial the same digits, but each group can have a different set of dialed digits. Using the technique to be described later, it is possible to terminate five lines dialing the same number to five different lines. Audible signals can

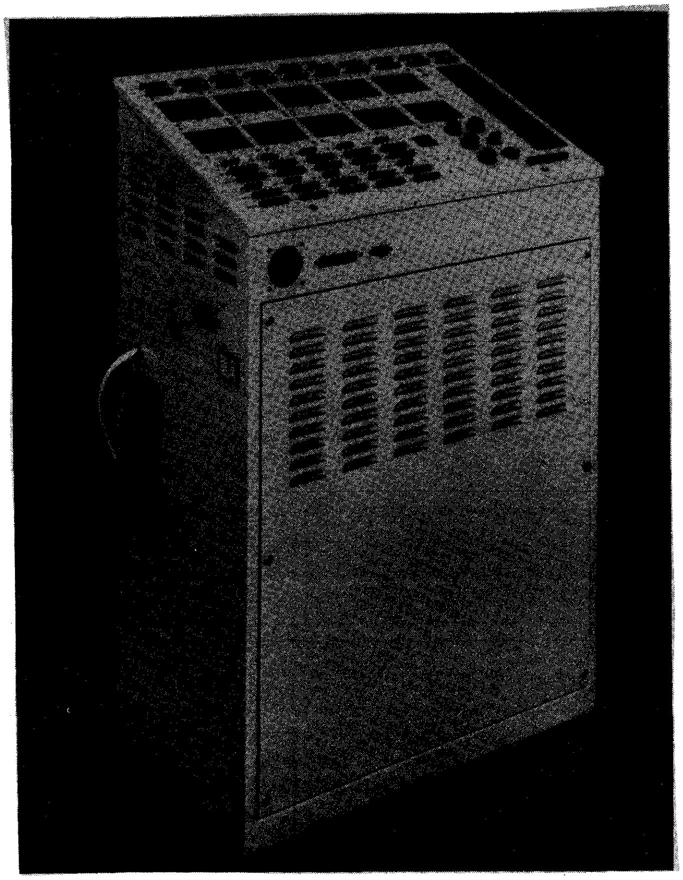


Figure 3—Load box

be monitored on a single line at any time through the use of the monitor amplifier and speaker furnished. Lamps are also provided for indicating the states of the lines, such as origination, dialing, and disconnect. Various timing adjustments can be made which determine when to start dialing, disconnect, etc. It should be pointed out that most test set actions are governed by time delays rather than system responses. For example, a set will start dialing after a preset delay following origination, whether dial tone is received or not.

The test set also provides termination for 50 lines. Each terminating line is equipped with a lamp, counter, and a circuit to detect ringing. Upon reception of a call, this circuit will trip ringing—simulating an answer back to the system, light the lamp and increment the counter. The circuit also applies a special tone to the terminating line for manual monitoring purposes. When this tone is received at the originating end, it verifies that a talking path is indeed established. The counter is used to determine the number of calls that are completed to talking.

The load box traffic tends to be more bunched than real customer traffic. Although staggered originations and disconnects are provided among the ten groups in a load box, the five lines in each group, nevertheless, will originate and disconnect simultaneously. Dialing for all 50 lines will be done at the same time. Therefore, the load presented to the system by the load boxes is more severe than one would encounter with real life traffic.

Techniques

A number of techniques are used to get around some of the constraints and problems associated with the test environment and the existing test equipment.

To simulate an outgoing call to another office and an incoming call from a distant office, a loop-around technique is employed. Under this case, a call is originated from a test line to an outgoing trunk. The output of this trunk is then fed back into No. 1 ESS by looping around the tip and ring conductors of the outgoing trunk to an incoming trunk. The system completes the loop by placing a terminating call to another test line in the office. Each loop-around call, therefore, consists of two calls, one outgoing and one incoming call. If a load box repeats its cycle every 36 seconds, then each test line can generate calls, either intraoffice or loop-around, at the rate of 100 per hour.

To terminate five lines in each group (dialing the same number) to five different lines, the speed calling technique is used. In No. 1 ESS, this is one of the new customer services provided. Speed calling permits a

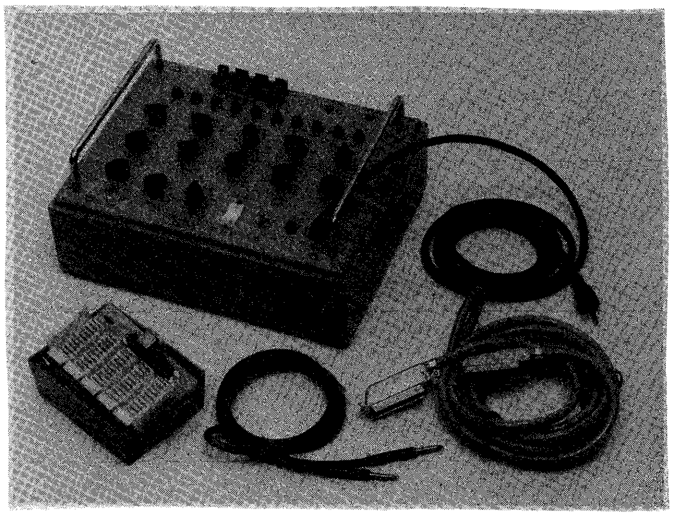


Figure 4—Sequencing unit

customer to place calls to one of a group of frequently called numbers by dialing an abbreviated code instead of the seven or more digits that would normally be required. The abbreviated code consists of an access code of 11 plus one or two digits depending upon the size of the abbreviated dial list.

With this method, five lines in the group are all assigned the speed calling feature, and each is given a different abbreviated dial list. Thus, all five lines dialing the same abbreviated dial code will place calls to five different terminating directory numbers. The calls can even be a mixture of intraoffice and interoffice calls. Also different types of signaling over different trunk groups can be selected for the interoffice calls. All this is accomplished by selecting the proper directory numbers for the abbreviated dial lists. The real-time consumption in processing a speed dialed call is about the same as a conventionally dialed call. The time saved in collecting the extra digits is consumed by the additional time required in translating the abbreviated code.

Eleven load boxes if simultaneously originating, dialing, outpulsing, and disconnecting, would place a very unrealistic load on the system. This is true even if enough service circuits existed in the office to permit such a test. Equipment limitations are major considerations in designing the load test. For example, the number of transmitters of a given type limits the number of interoffice calls that can be placed simultaneously at any given time. To stagger the operation of load boxes, a sequencing unit was designed (Figure 4). This unit generates start signals to load boxes in a fixed-time relationship. Therefore, it permits staggering of originations, dialing, and disconnects between load boxes. This results in a more evenly distributed load to the system, and simulates more closely the real-life traffic through the office.

With 11 load boxes operating at a 30-second cycle time, it was possible to generate 120,000 test calls per hour. The load box traffic mix was chosen to duplicate, as nearly as possible, the expected mix at the Portland office.

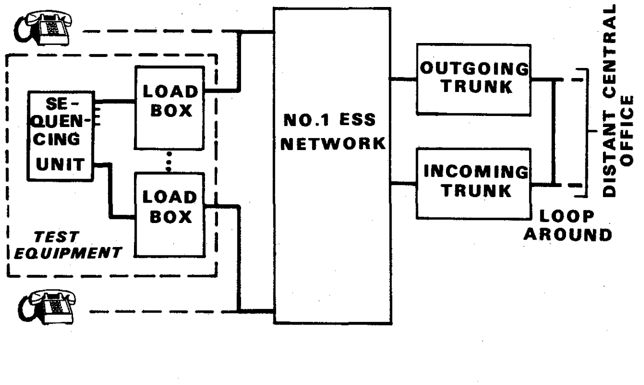


Figure 5—Load test arrangement

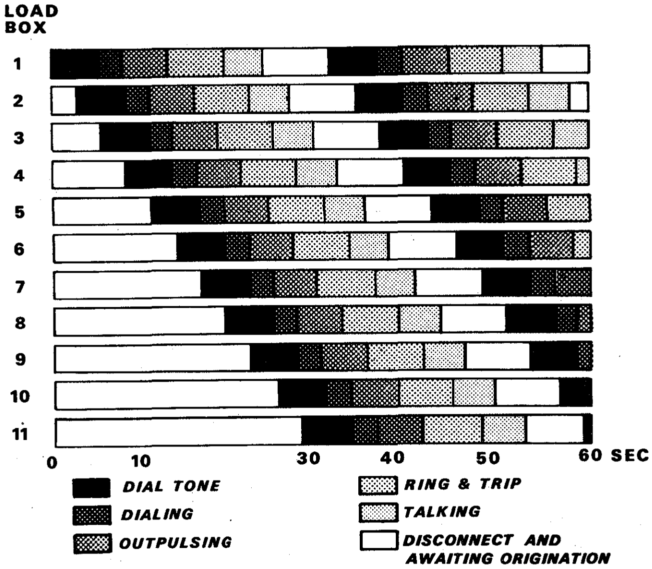


Figure 6—Load box timing chart

Figure 5 shows the load test arrangement. Figure 6 displays a simplified 30-second cycle timing chart for 11 load boxes. It gives the relationship between load boxes and the time allowed for the various phases of call processing: 5 seconds for dial tone, 3 seconds for abbreviated dialing, 5 seconds for outpulsing (transmitting signals over outgoing trunks), 6 seconds for the ring and ring trip, 2 to 4 seconds for talking, and 7 to 9 seconds for disconnect and awaiting origination. This type of timing chart is utilized to determine the maximum service circuit and trunk demand. Analysis of this nature is based on the concept that the load pattern is repetitive. It can be appreciated that accurate timing adjustments for load boxes and the sequencing unit are extremely important.

Test monitoring

There are many hardware and software performance indicators built within the No. 1 ESS. The system routinely prints out messages on various aspects of its well being.

One particular message, the quarter hour message, is of particular interest to us during the load test. It shows the total number of originating and incoming calls processed by the system in the preceding 15-minute period.

Dial tone speed test data are also included as a part of the message. To conduct a dial tone speed test, the system performs a routine test every 4 seconds (or 225 tests in 15 minutes). The test involves an origination from a random selected line. If dial tone is not detected within 3 seconds, a counter is incremented. Dial tone delay is an important indicator of the quality of service provided to the customers. The dial tone speed test data, therefore, is closely watched in the normal service of an ESS office. A high percentage of more than 3-second delays usually indicates an overload or some other trouble condition.

Another important data included in the quarter hour message is the number of the executive control or main program cycles. As traffic load builds up, the main program cycles get stretched out longer and longer. Consequently, the total number of main program cycles becomes less in a fixed length of time. The main program cycle rate, therefore, is an inverse function of machine load.

Early studies of the traffic data obtained from the then existing program have led to the use of 3500 as the minimum number of main program cycles in each 15-minute interval that can be tolerated while meeting all customer service requirements. The peak call capacity of the system, therefore, is the call rate which results in 3500 main program cycles per quarter hour.

The call completion rate, that is the number of calls completed to talking, can be derived from the load box counters. Since the ratio of originating calls and incoming calls is known from load box setups, the call completion rate can be calculated with reasonable accuracy from the quarter hour traffic data. This is a much easier method than the one which requires resetting and reading some 560 counters for each test.

TEST RESULTS

Call capacity

The observed call capacity follows closely but generally higher (about 5 percent) than the capacity predicted by the capacity-estimating ESS1CAP program.

For the Portland load box traffic mix (10 percent intraoffice, 45 percent incoming, 45 percent outgoing calls) the peak busy hour call capacity computed by the ESS1CAP is 108,000 calls at a main program cycle rate of 3500 per 15 minutes. ESS1CAP assumes that 85 percent of calls complete to talking and 15 percent to busy or no answer. The load test result shows 111,300 calls per hour at a main program cycle rate of 4767 per 15 minutes. There were no dial tone delays over 3 seconds during this test. This call completion rate was 99.8 percent. In real-time consumption, a completed call takes more machine cycles than a call to busy or no answer. Therefore, the overall results, in terms of the number of calls, main program cycles, and call completion rate, are considerably better than expected.

The highest load box traffic placed on the system was 118,080 calls per hour. For this test, the overload control was disabled with a program overwrite to avoid having the inputs rejected at a lower traffic level. At this load, 98.5 percent of calls completed to talking, and 1.5 percent to partial dial or recorder. A call will be routed to a reorder tone if a needed service circuit is not available. Partial dial in this case was caused by load box dialing before receiving dial tone. About 1.8 percent of calls encountered dial tone delays over 3 seconds. The main program cycles were 1092 in the 15-minute interval. The longest duration of a main program cycle was 5.5 seconds. This data was obtained through a program overwrite which prints out the number of various main program cycle durations in a special message. The system performed remarkably well even at such low number of main program cycles.

It is believed that the dial tone speed test failures and long main program cycle durations were caused primarily by load bunching of load boxes. Figure 7 shows a plot of the number of main program cycles per second versus elapsed time on one of the test runs. The load bunching is clearly evident. A Varian recorder (40 cm per second tape speed) was used in gathering this data.

Figure 8 shows some of the data on main program cycles versus traffic and ESS1CAP predictions. This figure applies to Portland office only.

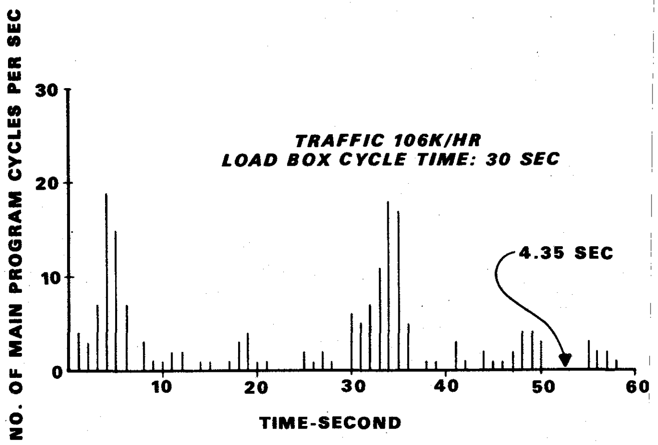


Figure 7—Distribution of main program cycles in load test—a random sample

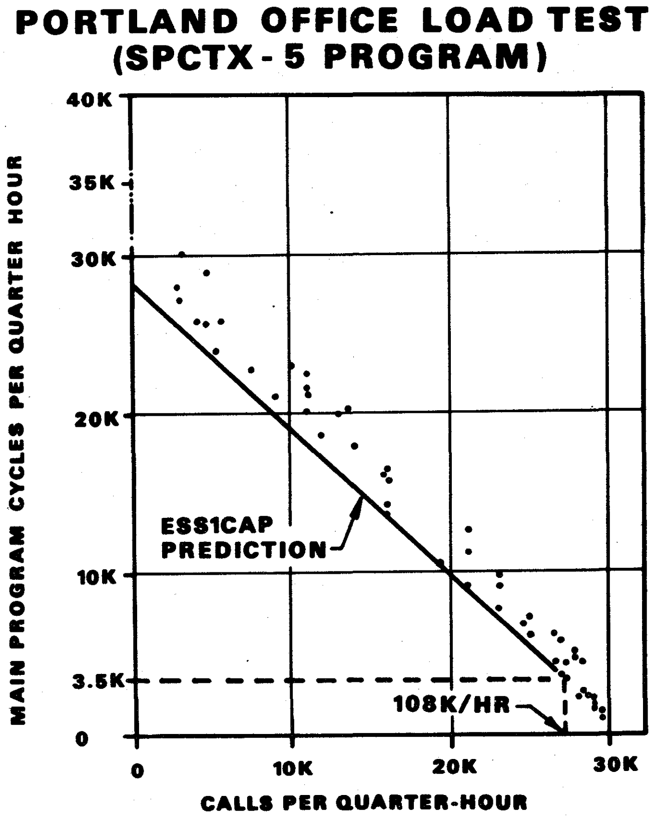


Figure 8—Number of main program cycles versus traffic

It should be pointed out that even though the number of calls in the load test matches closely the ESS1CAP predictions with similar number of main program cycles at very high traffic, the corresponding test load to the system is greater. This is because the call completion rate in the load test was higher.

Main program cycles

As mentioned earlier, to meet the service requirements, 3500 was used as the minimum number of main program cycles per quarter hour. The test result shows that at the Portland office, the system is capable of providing good service well below the 3500 minimum limit. When the system was processing calls at the rate of 116,300 calls per hour (8300 over ESS1CAP predicted peak capacity), the number of main program cycles was 2104 in a 15-minute interval. At this level of traffic, only 15.5 percent of originating calls encountered more than 3 seconds of dial tone delay. The service requirements allow 20 percent.

It appears that for the Portland office, the system can operate satisfactorily with 2500 main program cycles per quarter hour. The system not only would be able to meet, but also would exceed the service requirements. This corresponds to about 4 percent additional capacity over and above what has been achieved for the present program. Whether the lower main program cycle limit can be applied universally to all offices with the same program merits further investigation.

Overload control

The present overload control appears to be satisfactory.

A temporary program overwrite was installed so that the various overload parameters could be modified via teletype input messages. A series of load tests were made by varying this set of parameters. It was possible to clamp the quarter hour main program cycles in the general vicinity of any desired number during overload. In other words, we can limit the load to the system to any amount regardless of the service demands.

Based upon our experience at Portland, the existing values of overload parameters could be left as they are.

Program problems

The load test also provides maximum interactions for various segments of different call programs. Usually more program bugs will show up under heavy load, not because they are traffic dependent, but rather the conditions which lead to the bugs happen more frequently. During the entire 2-week load test period, only one call-affecting program problem was found, and this problem is truly traffic sensitive.

Many outgoing calls were lost during the early part of our test under heavy load. In ESS call processing, a certain timing is required to be done on outgoing trunks placed on a waiting list after each use. The program performs this timing on ten trunks every 200 milliseconds. In very high traffic, there were more trunks put on the list than could be taken off. Thus, not enough trunks were available to handle calls. A program change since has been made to correct this situation.

SUMMARY

A series of load tests have been made on the Bell System's No. 1 ESS latest program in a field office at Portland, Oregon. The results of these tests have validated the real-time improvements predicted. The call capacity estimate made by the ESS1CAP computer program is credible and conservative by about 5 percent. The system is capable of providing good service at the Portland office well below the main program cycle rate of 3500 per quarter hour. A 2500 figure is more realistic. This corresponds to a gain of an additional 4 percent capacity. Overload control appears to be satisfactory.

New improvements, primarily in programming rather than in hardware, are being made which will further increase the No. 1 ESS call capacity in the future.

REFERENCES

1 W KEISTER et al
No. 1 electronic switching system
Bell System Technical Journal Vol 43 Parts 1 and 2 September 1964

2 P NADOR S H TSIANG
Operational testing of software by means of simulation techniques
IEE Conference Publication No 52 International Conference on Switching Techniques for Telecommunication Networks London England April 21-25 1969

Minimum cost-reliable computer communication networks

by JOHN DEMERCADO

Ministry of Communications
Ottawa, Canada

INTRODUCTION

A designer of a computer-communications network must consider the reliability of a given network design as a function of its realization costs. Although there is an abundance of graph theoretic and queuing tools that have generated algorithms for the topological synthesis and analysis of large networks,[1,2,3] it is unfortunate that the reliability and cost dimensions of the problem have not been satisfactorily related.

In this paper a fast recursive algorithm[6] and elements of the theory of discrete Markov process[5] are combined to develop a new theory of reliability prediction for general networks whose nodes and links have constant failure and repair rates.

The methods presented are applicable to a large class of networks including computer-communication networks. The reliability theory as presented permits the time behavior of these networks to be rigorously treated.

In particular, methods are given for computing reliability functions[5] for the network. These functions given the probability that the network is in an acceptable state at time t; methods are also given for computing the moments of the first time that the network passes from given "acceptable" states to any arbitrary or specified "failure" states.

In the section on Preliminaries, a method is outlined for obtaining the transition probability matrix of a Markov chain that contains the per unit time probabilities of communication between each pair of nodes in the network.

In the section on Reliability Modelling these methods are applied to develop a reliability prediction model for any given network. An algorithm for minimum cost reliability modelling which delineates the computational procedure for using these results is then given. The recursive algorithm for computing the transition probability matrix for a general network is presented in the Appendix.

PRELIMINARIES

Let $|P_v|$ be the 2×2 transition probability matrix associated with node η_v of a n network η. That is

$$|P_v| = \begin{array}{c|c|c|} & A_v & F_v \\ \hline A_v & p_{a,a,v} & p_{a,f,v} \\ \hline F_v & p_{f,a,v} & p_{f,f,v} \\ \hline \end{array} \quad ; \quad v=1,\ldots,n \qquad (1)$$

where entry $p_{a,a,v}$ is the probability that node η_v which now operates successfully will operate again successfully one unit of time later. Node η_v is said to be in acceptable state A_v if it is operating successfully and failure state F_v if it is not. Similarly, let $|P_{vu}|$ be the 2×2 transition matrix associated with the link (uv) of the network η, that is:

$$|P_{uv}| = \begin{array}{c|c|c|} & A_{uv} & F_{uv} \\ \hline A_{uv} & p_{a,a,u,v} & p_{a,f,u,v} \\ \hline F_{uv} & p_{f,a,u,v} & p_{f,f,u,v} \\ \hline \end{array} \quad ; \quad (uv) \in \{L\} \qquad (2)$$

where $p_{a,f,u,v}$ is the probability that link (uv) which is now operating successfully will fail in the next period of time. The probability $p_{f,a,u,v}$ is the repair probability this is, the probability that if link (uv) is now failed it will be repaired in one unit of time. The link (uv) is said to be operating successfully when it is in state A_{uv} and unsuccessfully when it is in state F_{uv}.

The network η is thus specified by a set of n nodes,

$\eta_v; v=1, \ldots n$, denoted by $\{N\}$, a set of links $\{L\}$, and a set of matrices associated with these nodes and links.

Then for every node η_i and any other node η_j not directly connected to η_i by a single link, it is possible using the Algorithm given in the Appendix to compute the set of 2×2 matrices

$$|\chi_{ij}| = \begin{array}{c|c|c|} & A_{ij} & F_{ij} \\ \hline A_{ij} & \chi_{a,a,i,j} & \chi_{a,f,i,j} \\ \hline F_{ij} & \chi_{f,a,i,j} & \chi_{f,f,i,j} \\ \hline \end{array} \; ; \; j=1, \ldots, n \quad (3)$$

In equation (3), $|\chi_{ij}|$ is the one step transition matrix for the node pair η_i, η_j. In particular $\chi_{a,a,i,j}$ is the probability that there was communication between nodes η_i and η_j, at time t and that there will be communication at time $t+1$. The network is said to be in acceptable state A_{ij}, if the nodes η_i and η_j can communicate at time t, and in state F_{ij} otherwise.

For a n node network there are n such 2×2 matrices for each node η_i and it is possible to combine these into a transition probability matrix $|M_i|$ of dimension $2n\times2n$ for each node η_i as

$$|M_i| = \begin{array}{cc|c|c|c} & & A_{il} \ldots A_{in} & F_{il} \ldots F_{in} & \\ \hline A_i \to A_i & \left\{\begin{matrix} A_{il} \\ \vdots \\ A_{in} \end{matrix}\right. & |A_i| & |B_i| & \left.\right\} A_i \to F_i \\ \hline F_i \to A_i & \left\{\begin{matrix} F_{il} \\ \vdots \\ F_{in} \end{matrix}\right. & |D_i| & |C_i| & \left.\right\} F_i \to F_i \\ \hline \end{array} \quad (4)$$

Where A_i and F_i are the set of acceptable* and failed states associated with node η_i. They specify its operation with respect to the other nodes of the network. In general then the probabilistic behavior of the network can be characterized by the set $\{|M_i|\}$, $i=1, \ldots n$ of $2n\times2n$ matrices. The matrices $|A_i|$, $|B_i|$, $|C_i|$, $|D_i|$ are $n\times n$ square matrices which contain the one step transition probabilities. In particular

$|A_i|$, governs the transition from $A_i \to A_i$; $\forall_i=1, \ldots n$
$|B_i|$, governs the transition from $A_i \to F_i$; $\forall_i=1, \ldots n$
$|C_i|$, governs the transition from $F_i \to F_i$; $\forall_i=1, \ldots n$
$|D_i|$, governs the transition from $F_i \to A_i$; $\forall_i=1, \ldots n$

For the purpose of this paper the matrices $|C_i|$, $i=1, \ldots n$ will be considered as $n\times n$ unit matrices, corresponding to the case of distinct independent failures of individual nodes. The techniques presented could be modified for general $|C_i|$ matrices to include progressive degrees of failure, and dependence of failures of given nodes on other nodes. For the purposes of reliability modelling the matrices $|D_i|$ can be considered as $n\times n$ null matrices (all entries zero).

Furthermore, from the definitions given in Equations (3) and (4) it is readily apparent that the matrices $|A_i|$ and $|B_i|$ are diagonal matrices. This fact greatly simplifies computational procedures.

All the methods presented in this paper depend on the computation of the matrices shown in Equation (3). The recursive algorithm described in the appendix has been developed to calculate these transition probabilities for a general network.

RELIABILITY MODELLING

For each node of the network, a reliability function $R_i(t)$ can be defined as

$$R_i(t) = \text{Prob}\left[\begin{array}{l}\text{network } \eta \text{ is in every acceptable} \\ \text{state in } A_i \text{ at time } t\end{array}\right]$$

that is, node η_i is reliable provided it can communicate with all other nodes in the network at time t. Defining $\overline{S_i(o)}$ as the $(1\times2n)$ initial state vector for node η_i: Typically

$$\overline{S_i(o)} = |\underbrace{1, \ldots 1}_{n \text{ ones}},\ \underbrace{0 \ldots 0}_{n \text{ zeroes}}|$$

Let $\overline{S_i}(t)$ be the state vector at time t corresponding to node η_i, and the kth element of the vector $\overline{S_i}(t)$ be $S_{i,k}(t)$; then

$$R_i(t) = \prod_{k=1}^{n} S_{i,k}(t); \qquad i=1, \ldots n \qquad (5)$$

The product in (5) is over the set of acceptable states and the state probabilities satisfy[5]

$$\bar{S}_i(t) = \bar{S}_i(o)\ |M_i|^t$$

Transition failure probabi ities

Let $p_{ik}(t)$ be the probability that node η_i initially connected to node η_k is no longer connected to node η_k after t units of time. Let $|P_i(t)|$ the $n\times n$ matrix of these probabilities, then the following is true

$* A_i = \bigcup_{k=i}^{n} A_{ik}.$

Theorem 1

Consider a network η, with nodes $\eta_i \in \{N\}, i=1, \ldots, n$ and links $(ij) \in \{L\}$. Let the transition matrices for these individual nodes and links be $| P_v |$ for $\eta_v \in \{N\}$ and, $| P_{uv} |$ for $(uv) \in \{L\}$ respectively. Then

$$| P_i(t) | = | A_i |^{t-1} | B_i | + | P_i(t-1) |, \qquad (6)$$

where

$$| P_i(1) | \equiv | B_i |$$

Proof

This is a straightforward extension of Theorem 2 in Reference 5.

A special result of Theorem 1 is the following:

Corollary

The $n \times n$ matrix of the steady state probabilities defined as

$$\underset{t\to\infty}{\text{limit}}\, | P_i(t) | = | P_i | \qquad (7)$$

satisfy

$$| P_i | = || I | - | A_i ||^{-1} | B_i |, \qquad i=1, \ldots, n \qquad (8)$$

where $| I |$ is a $n \times n$ identity matrix. $| P_i |$ also will be a $n \times n$ identity matrix since all physical systems will ultimately fail with probability 1.

Proof

Equation (6) can be expanded as

$$| P_i(t) | = | I | + | A_i | + | A_i |^2 + \cdots + | A_i |^{t-1} | B_i | \quad (9)$$

therefore the limit in Equation (7) is the sum of the infinite Geometric series

$$| P_i | = \sum_{t=0}^{\infty} | A_i |^t | B_i |$$

which is (8) Q.E.D.

Moments of the first time to failure

The Reliability Modelling of the Network η is completed if in addition to the Equations (5), (6) and (8) it is possible to compute the moments of the first time that various types of disconnections occur in the network.

Closed form expression for these moments can be obtained for each of the nodes η_i in terms of the matrices $| A_i |$ and $| B_i |$. To obtain these expressions, define for node η_i the random variables ζ_{ik} as

$\zeta_{ik} \equiv$ "first time that node η_i is no longer connected to η_k."

Then $p_{ik}(t)$ is the probability distribution function of the random variable ζ_{ik} that is

$$p_{ik}(t) = \text{Prob}\ \{\zeta_{ik} = t\} \qquad (10)$$

Generating functions

Since ζ_{ik} is a discrete random variable its moments can be obtained from its generating function $g_{ik}(z)$ which is defined

$$g_{ik}(z) = \sum_{t=1}^{\infty} z^t p_{ik}(t) \qquad (11)$$

For each node $\eta_i \in \{N\}$ of the network η we obtain a matrix $| G_i(z) |$ of generating functions which can be written in matrix form as

$$| G_i(z) | = \sum_{t=1}^{\infty} z^t | P_i(t) |, \qquad i=1, \ldots, n. \quad (12)$$

Let $| \zeta_i(k) |$ be the $(n \times n)$ matrix of the kth moments, $k=1, 2, \ldots$ of the random variables $\{\zeta_{ij}\}$ for node η_i. Then:

$$| \zeta_i(k) | = \frac{d^k}{dz^k} | G_i(z) | \Bigg|_{z=1} ; \qquad \begin{cases} k=1, 2, \ldots \\ i=1, 2, \ldots, n \end{cases}$$

Using the Equation (6) it is possible to obtain a closed form expression for $| G_i(z) |$ and hence $| \zeta_i(k) |$ in terms of the matrices $| A_i |$ and $| B_i |$ without the need to evaluate infinite series of the form given in Equation (12). This result is given in Theorem 2.

Theorem 2

Let η be a n node network, with nodes $\eta_i \in \{N\}$, and links $(uv) \in \{L\}$ with corresponding one step node and link transition failure probabilities $| P_v |$ and $| P_{uv} |$. Then the generating functions $| G_i(z) |$ for the moments of the random variable ζ_{ij} "first time no connections exist between node η_i and node η_j" are given as

$$| G_i(z) | = \frac{z}{1-z} \Bigg| | I | - z | A_i | \Bigg|^{-1} | B_i |; \qquad i=1, \ldots, n$$

(14)

Proof

Substituting (6) into (12) and expanding gives

$$| G_i(z) | = \sum_{t=1}^{\infty} z^t | A_i |^{t-1} | B_i | + \sum_{t=1}^{\infty} z^t | P_i(t-1) | \quad (15)$$

the first term in (15) is $z || I | - z | A_i ||^{-1} | B_i |$ and the second is simply $z | G_i(z) |$. Q.E.D.

It can also be shown[5] that the moments $\zeta_i(k)$, $k=1$ the random variable ζ_i, where

$\zeta_i \equiv$ "first time that node η_i is disconnected from all other nodes"

are given by

$$\zeta_i(k) = \sum_{j=1}^{n} \zeta_{ij}(k) \qquad k=1, 2, \ldots \quad (16)$$

In particular for $k=1$, Equations (13) and (16) are the important mean times to first failure.

MINIMUM COST RELIABILITY MODELLING ALGORITHM

In general the problem facing the network designer is which equipment to use to realize a given network within a given cost range and with what reliability. There are many variations of this algorithm depending on which aspect of the network design problem is receiving the most emphasis. In the version indicated below emphasis will be placed on the problem of implementing the most reliable network that is below a certain cost.

Network costs

Let $\{C(N)\}$ and $\{C(L)\}$ be the cost matrices for the nodes and links of network η. That is the cost C of a realization of η, for a given topology and type of equipment is

$$C = \sum_{\eta_i \epsilon \{N\}} C(\eta_i) + \sum_{(uv) \in \{L\}} C[(uv)] \quad (17)$$

Algorithm

Given the network η, with nodes $\eta_i \in \{N\}$ and links $(ij) \in \{L\}$, and link and node cost transition matrices for r implementation options, that is given for each option r, the matrices

$$[| C^r(\eta_v) |; | C^r((uv)) |; | P_v{}^r |; | P_{uv}{}^r |] \qquad r=1, 2, \ldots$$

Find those options that will realize η with the best reliability and have cost less than or equal to some constant C.

$$\sum_{nv \epsilon \{N\}} C(\eta_v) + \sum_{(uv) \epsilon \{L\}} C(uv) \leq C$$

Step 1

Find the set of options whose realizations of η satisfy (17). If none then C is too low, and should be incremented by an amount ΔC and Step 1 repeated. When options are found go to Step 2.

Step 2

Use the Recursive Algorithm in the Appendix for each network node η_i and each of the options r, that satisfy the condition (17), to compute the general one step transition probabilities in Equation (3) and arrange these as matrix $| M_i |$ for the rth option

$$| M_i{}^r |; \quad i=1, \ldots n; \quad r=1, \ldots, h$$

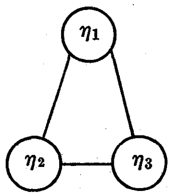
Go to Step 3.

Step 3

Let us assume that there are h such options, for each acceptable option r, $r=1, \ldots, h$, compute using the methods given in the paper $[R_i{}^r(t); | p_i{}^r(t) |; | \zeta_i{}^r(k) |; \overline{\zeta_i{}^r(k)}]$, $i=1, 2, \ldots, n$. The most reliable network is the one which has the "best" reliability function, longest mean time to failure, etc. Usually the designer can do the selection trade offs, by comparison of the above functions for the different options.

Example

Consider the network η with full duplex links



$\{N\} = \{\eta_1, \eta_2, \eta_3\}$, $\{L\} = \{(\eta_1\eta_2), (\eta_1\eta_3), (\eta_2\eta_3)\}$. The following options, can be used to construct this network. Furthermore the cost of the network realization should if possible not exceed $C=250$ units.

Data for Option 1

$$| C^1(N) | = | 100, 50, 50 |, | C^1(L) | = \begin{bmatrix} 0 & 5 & 6 \\ 5 & 0 & 7 \\ 6 & 7 & 0 \end{bmatrix}$$

$$| P_1^1 | = \begin{bmatrix} .90 & .05 \\ .05 & .95 \end{bmatrix},$$

$$| P_2^1 | = \begin{bmatrix} .92 & .08 \\ .07 & .93 \end{bmatrix},$$

$$| P_3^1 | = \begin{bmatrix} .89 & .11 \\ .06 & .94 \end{bmatrix}$$

$$| P_{12}^1 | = | P_{21}^1 | = \begin{bmatrix} .85 & .15 \\ .1 & .9 \end{bmatrix}$$

$$| P_{13}^1 | = | P_{31}^1 | = \begin{bmatrix} .9 & .1 \\ .20 & .8 \end{bmatrix}$$

$$| P_{23}^1 | = | P_{32}^1 | = \begin{bmatrix} .91 & .09 \\ .12 & .88 \end{bmatrix}$$

Data for Option 2

$$| C^2(N) | = | 75, 25, 75 |; | C^2(L) | = \begin{bmatrix} 0 & 6 & 5 \\ 6 & 0 & 8 \\ 5 & 8 & 0 \end{bmatrix}$$

$$| P_1^2 | = \begin{bmatrix} .90 & .1 \\ .12 & .88 \end{bmatrix},$$

$$| P_2^2 | = \begin{bmatrix} .85 & .15 \\ .2 & .8 \end{bmatrix},$$

$$| P_3^2 | = \begin{bmatrix} .9 & .1 \\ .04 & .96 \end{bmatrix}$$

$$| P_{12}^2 | = | P_{21}^2 | = \begin{bmatrix} .9 & .1 \\ .2 & .8 \end{bmatrix}$$

$$| P_{13}^2 | = | P_{31}^2 | \begin{bmatrix} .85 & .15 \\ .08 & .92 \end{bmatrix}$$

$$| P_{23}^2 | = | P_{32}^2 | = \begin{bmatrix} .88 & .12 \\ .14 & .86 \end{bmatrix}$$

Use of the Algorithm

Step 1

Using Equation (17) we find for these two options
Option 1: Cost 236 units $< C = 250$ units
Option 2: Cost 213 units $< C = 250$ units

∴ Both Option 1 and Option 2 must be considered as possibilities in realizing the network η.

Step 2

The recursive algorithm in the Appendix is now used to find general one step transition matrices for each node η_i, $i = 1, 2, 3$, for each of the two options $r = 1, 2$.

Option 1: node η_1

$$| M_1^1 | \equiv \begin{array}{c|ccc||ccc|} & A_{11} & A_{12} & A_{13} & F_{11} & F_{12} & F_{13} \\ \hline A_{11} & .90 & 0 & 0 & .10 & 0 & 0 \\ \hline A_{12} & 0 & .86 & 0 & 0 & .14 & 0 \\ \hline A_{13} & 0 & 0 & .88 & 0 & 0 & .12 \\ \hline \end{array}$$

$$\equiv \boxed{| A_1^1 |}\boxed{| B_1^1 |}$$

Option 1: node η_2

$$| M_2^1 | \equiv \begin{array}{c|ccc||ccc|} & A_{21} & A_{22} & A_{23} & F_{21} & F_{22} & F_{23} \\ \hline A_{21} & .90 & 0 & 0 & .10 & 0 & 0 \\ \hline A_{22} & 0 & .80 & 0 & 0 & .20 & 0 \\ \hline A_{23} & 0 & 0 & .85 & 0 & 0 & .15 \\ \hline \end{array}$$

$$\equiv \boxed{| A_3^1 |}\boxed{| B_3^1 |}$$

Option 1: node η_3

$$| M_3^1 | \equiv \begin{array}{c|ccc||ccc|} & A_{31} & A_{32} & A_{33} & F_{31} & F_{32} & F_{33} \\ \hline A_{31} & .88 & 0 & 0 & .12 & 0 & 0 \\ \hline A_{32} & 0 & .85 & 0 & 0 & .15 & 0 \\ \hline A_{33} & 0 & 0 & .75 & 0 & 0 & .25 \\ \hline \end{array}$$

$$\equiv \boxed{| A_3^1 |}\boxed{| B_3^1 |}$$

Option 2: node η_1

$$\begin{array}{r|ccc||ccc|}
 & A_{11} & A_{12} & A_{13} & F_{11} & F_{12} & F_{13} \\ \hline
A_{11} & .9 & 0 & 0 & .1 & 0 & 0 \\ \hline
|M_1^2| \equiv A_{12} & 0 & .88 & 0 & 0 & .12 & 0 \\ \hline
A_{13} & 0 & 0 & .82 & 0 & 0 & .18 \\ \hline
\end{array}$$

$$\equiv \boxed{|A_1^2|}\boxed{|B_1^2|}$$

Option 2: node η_2

$$\begin{array}{r|ccc||ccc|}
 & A_{21} & A_{22} & A_{23} & F_{21} & F_{22} & F_{23} \\ \hline
A_{21} & .91 & 0 & 0 & .09 & 0 & 0 \\ \hline
|M_2^2| \equiv A_{22} & 0 & .83 & 0 & 0 & .17 & 0 \\ \hline
A_{23} & 0 & 0 & .80 & 0 & 0 & .2 \\ \hline
\end{array}$$

$$\equiv \boxed{|A_2^2|}\boxed{|B_2^2|}$$

Option 2: node η_3

$$\begin{array}{r|ccc||ccc|}
 & A_{31} & A_{32} & A_{33} & F_{31} & F_{32} & F_{33} \\ \hline
A_{31} & .87 & 0 & 0 & .13 & 0 & 0 \\ \hline
|M_3^2| \equiv A_{32} & 0 & .79 & 0 & 0 & .21 & 0 \\ \hline
A_{33} & 0 & 0 & .81 & 0 & 0 & .19 \\ \hline
\end{array}$$

$$\equiv \boxed{|A_3^2|}\boxed{|B_3^2|}$$

Step 3

The methods given in the Paper can now be used to compute the reliability function, mean times to failure, etc., for each of the network realizations using option 1 and option 2. Computation indicates that **option 2** will yield better performance than **option 1** even though it costs less.

ACKNOWLEDGMENTS

The author would like to thank Mr. Nicholas Spyratos for his assistance in the preparation of this paper, and his secretary Miss Gail Widdicombe for expertly producing the manuscript.

REFERENCES

1 H FRANK I FRISCH
Communications, transmission and transportation networks
Addison Wesley 1970

2 J DEMERCADO N SPYRATOS
The synthesis of non flow redundant computer communications networks
Proceedings Brooklyn Polytechnic Symposium on Computer-Communications and Teletraffic N Y C
April 1972

3 J DEMERCADO K TOTH
The synthesis of computer communication networks
Department of Communications Report June 1972 available from Library Department of Communications 100 Metcalfe Street Ottawa Ontario

4 E HANSLER
A fast recursive algorithm to calculate the reliability of a communication network
IEEE Transactions on Communications Vol COM-20 No 3 June 1972 pp 637-640

5 J DEMERCADO
Reliability prediction studies of complex systems having many failed states
IEEE Transactions on Reliability pp 223-230 Vol R-20 No 4 November 1971

6 J DEMERCADO N SPYRATOS
Recursive algorithms for stochastic networks
(To appear)

APPENDIX

In this appendix an algorithm[6] is presented for determining the probability of disconnection between any two nodes of a general communication network with failing links and nodes. This algorithm offers considerable computational savings compared to a recent algorithm by Hansler.[5]

Notation

The following symbols are used:

$p_{a,f,i}$ is the probability that node η_i operates, at time t, but fails at time $t+1$.

$p_{a,f,i,j}$ is the probability that the link (ij) operates at time t but fails at time $t+1$.

$\chi_{a,f,i,j}$ is the probability* that node η_i can communicate with node η_j at time t but there is no communication at time $t+1$.

d_i denotes the degree of node η_i. That is the number of linkcs onnected ot this node.

* $\chi_{a,\ f,\ i,\ j}$ should be identified as the transition probabilities given in equation (3).

N_i denotes the set of nodes at the ends of the d_i links having η_i as a common terminal node.

To simplify the notation we suppose that the network has n nodes and the probability $\chi_{a,f,l,n}$ is to be calculated.

The recursive algorithm

Define the following sets:

$Y_1 = \{x, y\}$ where x means failure and y operation of node η_1.

$A = \{\eta_i \in N / \eta_i$ cannot communicate with η_n at time $t+1\}$

$B = \{\eta_i \in N /$the link (i, j) fails at time $t+1\}$

Now the space $Y_1 \times P(N_1) \times P(N_1)$, where $P(N_1)$ denotes the power set of N_1, is clearly the sample space on which the failure events for the network must be identified.

Suppose that σ is the probability measure on the sample space, γ the probability measure on $P(N_1) \times P(N_1)$ and μ the probability measure on $P(N_1)$.

The events of failure for the network belong to one of the following two classes of events:

$$F_1 = \{(x, A, B)/A, B \in P(N_1)\}$$

$$F_2 = \{(y, A, B)/AUB = N_1\}$$

Therefore,

$$\begin{aligned}
\chi_{a,f,l,n} &= \sum_{F_1} \sigma[(x, A, B)] + \sum_{F_2} \sigma[(y, A, B)] \\
&= \sum_{A,\ B \epsilon P(N_1)} p_{a,f,i} \cdot \gamma[(A, B)] \\
&\quad + \sum_{A=N-B} (1-p_{a,f,i}) \gamma[(A, B)] \\
&= p_{a,f,l} \sum_{A,B \epsilon P(N_1)} \gamma[(A, B)] \\
&\quad + (1-p_{a,f,l}) \sum_{A=N-B} \mu(A) \mu(B) \\
&= p_{a,f,l} + (1-p_{a,f,l}) \sum_{A=N-B} \prod_{N-B} \chi_{a,f,i,n}
\end{aligned}$$

$$\prod_{B} (1-\chi_{a,f,i,n}) \prod_{B} \chi_{a,f,l,i} \prod_{N-B} (1-p_{a,f,l,i})$$

$$\begin{aligned}
&= p_{a,f,l} + (1-p_{a,f,l}) \sum_{A=N-B} \prod_{B} p_{a,f,i,n} (1-\chi_{a,f,i,n}) \\
&\quad \times \prod_{N-B} \chi_{a,f,i,n} (1-p_{a,f,l,i})
\end{aligned}$$

The last formula is a recursive one since $\chi_{a,f,i,n}$ is the probability of disconnection between η_i and η_n but in a simpler network.

Comments

Since N_1 contains d_1 elements, the various ways we can set $A = N - B$ are, in all 2^{d_1} and, therefore the number of terms in this formula is $1 + 2^{d_1}$. On the other hand, Hansler's recursive formula uses $1 + 2^{2d_1}$ terms. Therefore the computational savings of the present method are $(1+2^{2d_1}) - (1+2^{d_1}) = 2^{2d_1} - 2^{d_1}$.

The Control Data® Star-100 file storage station

by G. S. CHRISTENSEN and P. D. JONES

Control Data Corporation
St. Paul, Minnesota

INTRODUCTION

Successful experience with the Control Data® 6000[1] and 7000[2] computer series has led to implementing improved concepts[3,4,5] of distributed computing in the STAR-100 computer system. In the STAR system different computing functions have been physically separated from one another. Each computing function is performed by an independent system unit which possesses its own processing logic and memory. Thus each is performed in its own right in an optimal manner.

STAR-100 computer[6] is a high speed processor capable of producing 100 million results (from a multiply operation, for instance) per second in its 4 or 8 million byte core memory. STAR itself cannot perform data input/output, this is performed by input/output units called stations which have channel interfaces to STAR. A station consists primarily of a mini-computer specially designed for data handling. The STAR design is thus simplified by not having to contain device interfaces; this modularity is important in the design of large computer systems.[7] Also the processor overhead of driving peripheral devices is relegated to the stations thus freeing STAR for additional user computation. Experience in several hundred Control Data® 6000 computer sites has shown it impossible to operate very high speed computers efficiently without distributing peripheral functions. As well as distributing the peripheral device drivers in STAR it has been found possible to perform system functions, such as file management, in the stations. So far 9 different STAR station types have been identified and built, these include: maintenance and monitoring, paging, storage, media (tape and disk), unit record, communication, display/edit, graphic and service. These contain the same basic hardware and software but vary at the device controller and system software interface level. The service station is a key station in that it manages the system resources and provides fan-out to the second level stations.

Operating system functions are thus distributed in a manner which closely follows the distribution of the hardware. The connecting links between the distributed operating system functions are controlled by a set of system messages and message handling is a key factor in efficient operation of the system.

The choice of where each operating function should be located is often self-evident, although a few functions are assumed to be movable from one element to another. Any final decision regarding function locations may depend on experience with particular work loads. In general each operating function is located closest to the resource being used and may be local or remote to the STAR processor. This provides modularity of both hardware and software and such advantages as:

- independence from other units, particularly in the areas of non-propagation of errors throughout the system and more immediate action on fault conditions.
- capability to be independently maintained.
- easier replacement of future new hardware or software parts.
- easier addition of new types of stations.

Figure 1 illustrates the layout of a large STAR system showing the connections between the various stations.

A STAR central processor with its immediate storage is simply another station within the system—a data processing station—and in no way does it have any extra authority. It does, however, have two stations

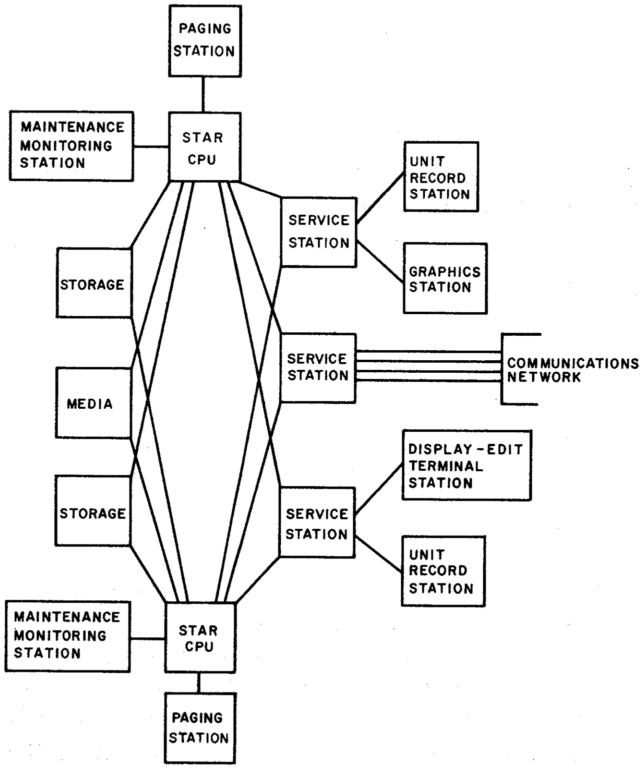


Figure 1—STAR system showing station connections

fairly intimately connected, the paging station and the maintenance/monitoring station. The paging station, under control of the hardware virtual page mechanism and the operating system, provides temporary storage for programs exceeding the available core space. The maintenance station, besides its functions of off-line and on-line fault diagnosis/repair and preventive checking, has the capacity to collect detailed information about STAR's performance.

The data management function is performed by programs executed within the central processor. These functions include merge, sort, select, scan, append, extract and insert. The data manager in turn exploits the storage station via message commands. This paper describes the storage station which manages the storage and retrieval of working and archival files.

STATION HARDWARE

The hardware used to implement the distributed computing concept in STAR is designated as various classes of input/output stations. Each Star channel terminates at a station with a common interface. The station (Figure 2) consists of an SCU (Station Control Unit) and an SBU (Station Buffer Unit).

The SCU consists of a mini-computer, display/keyboard, small drum and channel interfaces which exist with power supplies, cooling fan and operator panel in one cabinet. The mini-computer has an instruction set which caters to bit and byte manipulation. It contains 8K (K=1024) 8-bit bytes, expandable to 16K of 1.1 microsecond core memory. There is a 200 nanosecond version of the same meory but the 1 MIP (million instructions per second) rate of the computer is adequate for most present applications. The drum has an average access time of 17 milliseconds and a capacity of approximatly 80,000 bytes. It is used as a store for program overlays and also as a refresh memory for the display console.

The mini-computer (or buffer controller) provides a single, parallel-block transfer channel with hardware control for high speed data transfer. Its maximum rate is one 16-bit word plus two parity bits per memory cycle, 1.1 microseconds. The buffer controller also provides up to 512 normal channel bits for lower speed data transfer and device and station control. These bits are organized into 16 input channels and 16 output channels with 16 bits in each channel. Their use is determined by the individual peripheral devices on the station. The normal channel bits of the buffer controller provide the primary mechanism for control of the other station elements and the attached devices. A direct

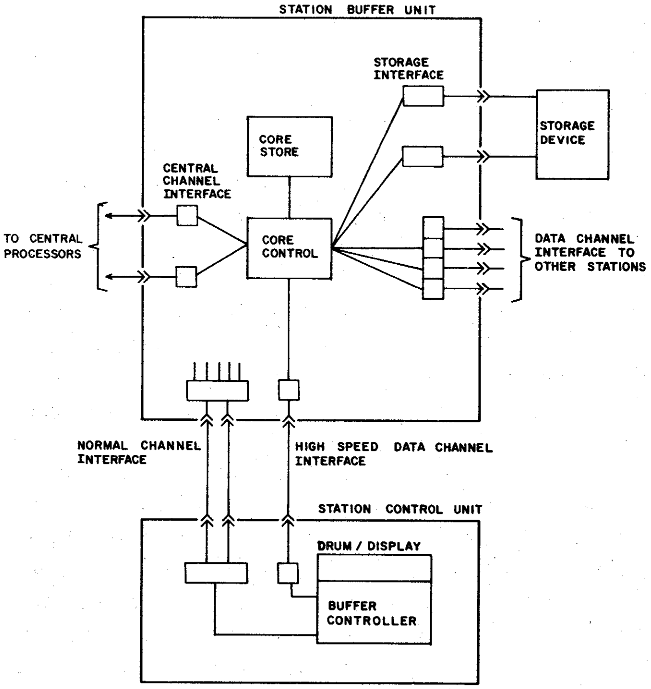


Figure 2—STAR station

interface of normal channel bits is provided between the SCU and the SBU (Figure 2).

The SBU consists of up to 64K bytes of memory organized in eight interleaved banks of 8K bytes each. Each bank has a memory cycle of 1.1 microseconds with a maximum bandwidth of 14 million bytes per second. Storage control logic provides for 12 independent channel accesses. The SBU is always associated with a controlling SCU. The general function of the SBU is to provide intermediate buffering of data, fan in/out from one STAR channel to many other station channels and working storage for the station. The interfaces to attached devices are contained in the SBU.

The following features of the SBU and its interfaces are important to its application and performance as a storage control mechanism.

- The high bandwidth allows simultaneous transfer of a number of storage devices into the SBU. The CDC 844 disk pack, for instance, has a transfer rate of approximately 1 million bytes per second compared with the SBU bandwidth of 14 million bytes per second.
- Device control operations such as connecting, addressing, and status are accomplished directly from the SCU over the buffer controller normal channel to the SBU device interfaces. This provides direct, detailed control of the devices.
- Actual data transfer between a storage device and SBU takes place automatically under control of the SBU device interface hardware. This frees the SCU during SBU data transfers.
- The SCU can directly access STAR storage via normal channel bits and the SBU interface. This mode is advantageous for message transfer and queue control.
- The SBU device interfaces are capable of stacking (queueing) functions and data transfer specifications. This allows maximum performance of the devices while relieving the SCU of having to intervene during brief, critical events such as crossing of intersector gaps.
- The SBU device interfaces have the capability of chaining SBU memory areas creating a contiguous data stream to a storage device from several SBU memory areas. This is used to automatically assemble and disassemble sync pattern and header information with the data block.
- All data is stored in fixed length blocks of 4096 bytes.

Storage station software

Tasks are communicated to the storage station via system messages. Each message selects a specific task and is handled by an SCU routine referred to as a task overlay. The task overlay contains the control code necessary to accomplish the task by calling various station subroutines and device drivers.

Associated with each device attached to a station is a device software driver in the SCU. This is a specialized routine which actually drives the devices through the SBU hardware interfaces. The other station routines communicate with the drivers through a driver parameter table and a driver-maintained status table. One status table exists for each device.

In addition to the device drivers other station subroutines are associated with station resource management and utility functions. Examples of these are:

- Rent buffer space in SCU
- Rent block in SBU
- Transfer SBU/SCU data
- Transfer CPU data
- Hash file name

Each station contains a standard program referred to as the nucleus or monitor. It contains a set of simple diagnostic routines known as quick-look diagnostics, a system autoload program, driver programs for the microdrum and for the keyboard associated with the character display, programs to manage the microdrum overlay mechanism, and the main control and organizational program.

The SCU microdrum holds a copy of all station software. The SCU operates under one of four different systems. These systems are allocated as follows:

1. Microdrum loader system
2. Run system (normal case)
3. Diagnostic system
4. Off-line system

The system is selected at start-up of SCU programs. The selection of a system causes linking of all routines associated with the system via scanner and overlay tables. When running, a given system contains the operating portion of the nucleus (the system selection and set-up routines are discarded to be called again from the microdrum for a new autoload) and specified routines fixed in core. The remaining routines are called when required from the microdrum. Calling a routine is accomplished through an overlay table which contains the core address of the called routine or the address of a routine which reads it into a core area available for temporary overlay and buffers. All routines associated with a system are thus directly accessible yet only the most active routines reside dynamically in the SCU core memory.

The scanner is the idle loop of the nucleus. The primary purpose of the scanner is to map normal channel data signals to overlay programs based on priority and logical selection, thereby providing a low overhead mechanism for handling asynchronous external events. The external events (such as channel flags, microdrum busy, or input read signals) are presented to the scanner program via one or more normal channels. Associated with each channel are two logical selection words, the ENABLE mask, and the STATE mask. The channel data is exclusive or'ed with the state mask in order to select the appropriate signal polarities, and then matched against the enable mask. Any bits that are now set represent selected channel events in the desired state. These bits are scanned from left to right and the first bit found set is used to enter the overlay program associated with that bit. If all bits are zero, the scanner moves on to the next channel and repeats the procedure. One or more memory words are used to initiate internal events via the scanner. In this case, the memory words rather than the channels represent the raw input to the scanner. In a typical station, the scanner cycles through two normal channels and two memory words.

A detailed error handling and maintenance system is provided in the stations. Abnormal conditions in the operation of a device cause the device driver to exit to an associated error handling routine. This routine handles retries and error logging. It operates in conjunction with a device monitor routine which is used to set the parameters for a device, such as number of retries, turning device off to system, and breakpointing in the driver. A maintenance information system provides an English translation of the driver parameter tables and the device status tables on the SCU display and provides operator access to control the device operation via the device monitor.

Included in the maintenance system is the capability to run diagnostics and utilities associated with a device. These tests are controlled using the device driver, parameter table, and status tables and may be run in conjunction with the system operation on other station devices.

FILE SYSTEM

The file system described here exists totally within the storage station and is independent of any particular processor station, network configuration or storage device type. Creation, maintenance, recovery, access, security, storage layout, accountancy data, and performance statistics are all managed within the station.

The station file system is implemented as a set of task overlays. Each overlay is associated with a specific system message and provides the coordination necessary to accomplish the system task using the station device drivers and subroutines. Each message has a separate overlay to process it. If the message occurs frequently, the overlay remains in SCU core; otherwise, it is called in from the microdrum when it is needed.

Active file index

All the file messages are listed in the Storage Station Messages section. A file is simply a collection of stored bits, which has a descriptor and can be operated on by a set of file functions. No file function is processed until the file is first opened, and the last file function must always be a close function. In the open message, identification of a file is by file name (File Name Section). For other messages, identification of a particular file is by its active file index, the index of the file entry in the active file table (Figure 3). The file index is assigned by the storage station and returned to STAR in response to the open message. The advantage of this arrangement is that the majority of file messages use a 16-bit identifier rather than a variable length string of characters which could be quite long. By maintaining active-file information in core storage, access validation and transformation between logical (file page) and physical block locations is normally accomplished with negligible overhead and without introduction of superfluous input-output operations.

The size of each active file table entry is 8 characters (Figure 3). Initially, one SBU block of 4096 characters is devoted to the active file table, allowing 512 open files at any one time. This can be easily expanded if required. If the file is noncontiguous, read/write of file pages which are not in the first contiguous section require an access to the storage map in the file descriptor. One could trade the number of open files allowed for fewer open files with each entry containing the map of more than one file section.

O/1	F	M	U	S	N	
1	15	8	8	16	16	BITS

O/1	=	free/used flag
F	=	description pointer
M	=	access mode
U	=	unit number
S	=	starting address of file on device
N	=	number of blocks contiguous to S

Figure 3—Active file table entry format

File descriptor (catalog entry)

Each file has a descriptor which describes the file as seen by the system. The descriptor (Figure 4) consists of 8 sections: Header, characteristics, name, storage map, access map, activity map, and two free sections reserved for later use.

The set of descriptors for those files occupying a particular storage unit is itself part of a file and may be processed like any other file; it is called the descriptor file or catalog. This catalog may or may not be on the same storage media as the files it describes. Normally, removable media contain their own catalog files, but these may be copied elsewhere on mounting.

The size of an individual descriptor is variable in modules of 256 bytes up to a maximum of 4096 bytes. Initially, just one module (256 bytes) is used for each descriptor.

As an example the Control Data® 844 disk pack at present has the following layout.

Blocks 0, 1	Pack Label	Pack Catalog File
Blocks 2, 3	Free Storage Map	
Blocks 4 through 67	Descriptor Modules (1024)	
Blocks 68 through 23,027	Data Files	

To facilitate processing in the SCU, the descriptor proper is kept reasonably small, but the sections can have pointers to overflow areas and these may be of any length. The space allocated for the catalog is also variable. Initially 64 blocks of 4096 characters are used providing 1024 files per storage unit.

The allocation of a descriptor module to a newly created file is done either by the use of a free space map for the modules or by a hashing algorithm. To locate a file descriptor, the file name is hashed to locate a bucket in a hash table which contains entries of file names and pointers to their descriptors. This hash table is re-created (say at autoload) so that the system is not tied to any one hashing algorithm. The hash table may itself become quite long and is kept on the storage unit with the files or some associated storage device. An alternate implementation simply hashes directly to the descriptor module. If the file name does not match the name in that module, a search is made of the surrounding modules in that block. It is to be emphasized that normally the descriptor is only referenced on the open and close functions. All read/write file pages reference the active file table which is in SBU core.

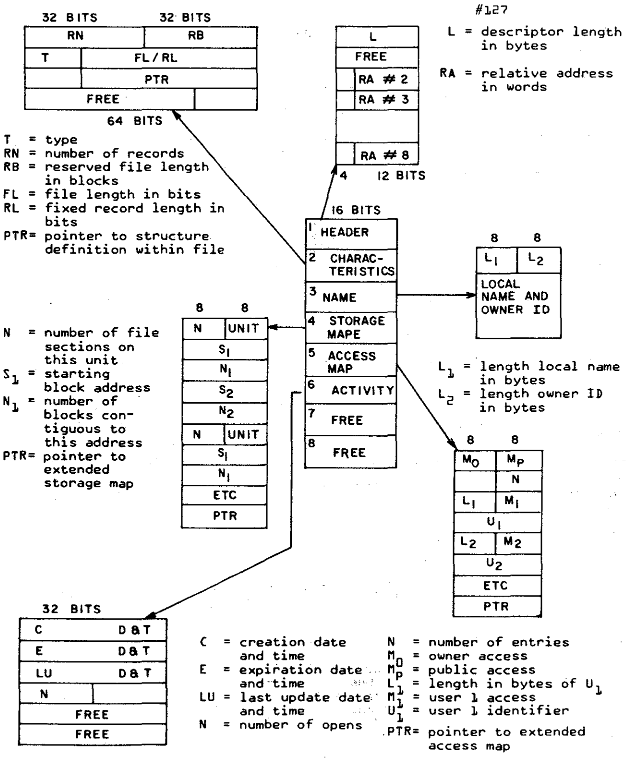

32 BITS
32 BITS
RN
RB
T
FL/RL
PTR
FREE
64 BITS
T = type
RN = number of records
RB = reserved file length in blocks
FL = file length in bits
RL = fixed record length in bits
PTR= pointer to structure definition within file
L
FREE
RA # 2
RA # 3
RA # 8
4
12 BITS
#127
L = descriptor length in bytes
RA = relative address in words
16 BITS
1 HEADER
2 CHARACTERISTICS
3 NAME
4 STORAGE MAPE
5 ACCESS MAP
6 ACTIVITY
7 FREE
8 FREE
8
8
L1
L2
LOCAL NAME AND OWNER ID
L1 = length local name in bytes
L2 = length owner ID in bytes
N = number of file sections on this unit
S1 = starting block address
N1 = number of blocks contiguous to this address
PTR= pointer to extended storage map
8
8
N
UNIT
S1
N1
S2
N2
N
UNIT
S1
N1
ETC
PTR
8
8
MO
MP
N
L1
M1
U1
L2
M2
U2
ETC
PTR
32 BITS
C
D & T
E
D & T
LU
D & T
N
FREE
FREE
C = creation date and time
E = expiration date and time
LU = last update date and time
N = number of opens
N = number of entries
M0 = owner access
Mp = public access
L1 = length in bytes of U1
M1 = user 1 access
U1 = user 1 identifier
PTR= pointer to extended access map

Figure 4—File descriptor format

Storage map section

The storage map (Figure 4) allows for a storage system to be divided into 256 units, each with a capacity of 65,536 blocks (2^{28} bytes: approximately 268 million). A variation on this scheme is being implemented which has 32-bit field lengths for block addresses and number of blocks contiguous to an address. This will cater for larger storage systems with capacity up to 2^{32} (approximately 4 billion) blocks or 2^{44} (16 trillion) bytes.

Characteristics section

The characteristic section of the descriptor is shown in Figure 4; the different file types are undefined (0), ASCII coded delimited (1), AS CII coded fixed (2), binary STAR (3), binary fixed (4), foreign delimited (5), foreign fixed (6), virtual memory (7), drop (8), labeled (9), multiple volume (10), incomplete (11), temporary/permanent (12), input (13), and output (14).

Types 1 through 6 categorize file types according to their internal coding. The exact definition is not important but it should be noted that types 3 through 6 have an associated record map which describes the record structure of the file. A virtual memory file has a virtual address associated with each file page. The drop file is similar to the virtual memory file, it is a frozen image of an executing job which has been suspended for some reason together with the virtual address list and current program status information. The labeled file is one that has a label (somewhat similar to the file descriptor) within the file. These last three types use a pointer address to locate the relevant structural information within the file. The multiple volume/unit file is one that is spread over a number of storage units; yet, it is logically one file. An incomplete file is one upon which, although incomplete, processing begins; such is the case when processing begins after only a portion of tape is spooled onto a disk. No doubt other file types will be added, but these provide sufficient categorization for the present.

File name section

Perhaps the most important thing about a file is its name. It is that which identifies it uniquely and which must be used to open the file before it can be processed. The name consists of two parts, a local name followed by an owner identifier. Each part consists of a variable length string of characters (the ASCII alphanumeric set plus −$#). The parts are separated by the ASCII space character. Certain characters are reserved for special use within file names: *, /, ., &, |, and ?. The period character . for instance, is used to indicate some hierarchical structure within the name.

The file system is not normally concerned with the internal structure of either the local name or owner identifier, who gave this name or identifier, or where it came from. Essentially the name is used to locate the descriptor.

Example of File Name MATRIX J249

4D4154524958204A323439 {ASCII hexadecimal} notation}

local name separator owner identifier

Storage layout section

The storage layout of a file varies with the particular storage device but the goal in each case is the same, that is, to organize file storage in a manner which does not deter high-performance of expected access requests. A large block of data, stored as 128 consecutive physical blocks on a Control Data® 817 disk requires a little over a tenth of a second for transferring its half million bytes; stored differently, its transfer could take up to 10 seconds. The allocation and layout of a file are governed by a RENT/STORE routine which can be replaced or modified in order to implement more elaborate policies. This routine normally tries to allocate the desired number of blocks in a contiguous fashion; if this is not possible it will allocate the total space on as few large sections as possible.

The map of the disk file is a vector. Each element of the vector is a storage location and a number indicating how many blocks are contiguous to the location. As many contiguous sections as possible are represented in the descriptor proper and the rest are kept in an overflow area.

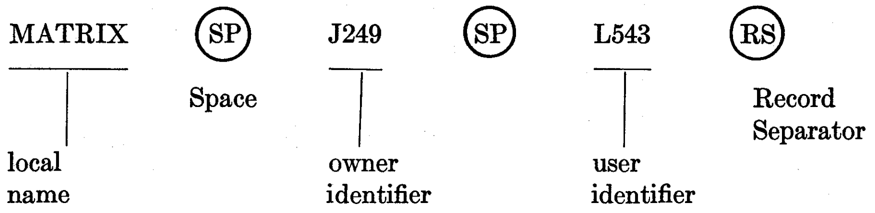
Access security section

Every time an OPEN operation is requested through a storage station, the access rights of the user are checked against the access map in the file descriptor. The open function has an owner identifier and a user identifier catenated to the local file name and terminates with the ASCII record separator code. If the user and the owner are the same person, the user identifier may be omitted. If the access is not permitted, an invalid access response is returned to the message sender. For the other file messages, validity of the operation is checked against the mode stored with the file entry in the active file table. Initial file access mode is one of four:

- Cannot delete
- Cannot alter access modes
- Cannot write
- Cannot read

The access modes for the different users are set or modified by access mode messages to the station. The default option on creation of a file is that the owner has open access and the public has no access. The file system again is not concerned with the internal structure of the user identifier; it is simply a variable length string of characters, and in fact, could be an agreed upon group name rather than an individual user identifier.

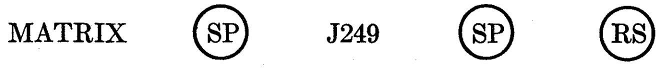
Example of File Identifier



MATRIX SP J249 SP L543 RS

local name — Space — owner identifier — user identifier — Record Separator

If the user is the owner, then this can reduce to



MATRIX SP J249 SP RS

Multiple stations

A typical STAR installation might include two STAR processors supported by a number of storage stations each having different storage devices attached. Such a system exists and is in experimental operation. It is possible for a user to specify on opening a file its location; if this is not done STAR sends messages to all storage stations listed in its directory. The station where the file exists opens the file and makes the appropriate response which STAR keeps till the file is closed. The other stations return a "not found" response.

At present on "create and open" the user must specify the storage station where the file is to be created but need not specify the device on that station unless he wished to do so. If a file of the same name already exists on the station it will be deleted if it is a "temporary" file and the new one will be created; otherwise, if it is a "permanent" file an "already exists" response will be returned to STAR. Files may be shared between different users and two STAR processors providing they are open for read only access. The station has no difficulty returning responses and data to the correct STAR processor since it is identified by its zip-code in the message header.

File system extensions

The basic file system can be extended to provide specific features. The basic file system and these extensions are expected to provide a very complete, stand-alone storage system.

- Automatic mounting—(packs cartridges, tapes, etc.)—Standard ASCII labels, automatic allocation of drives, and the mounting and dismounting with label validation.
- Multivolume files—Allowing a file to spread itself over a number of units.
- Archival file directory—One archival file directory for all files, on-line and off-line.
- Structured file name and owner/user identifiers—Structured names and identifiers linking files of a given class into a more complex access mechanism.
- Shared access security—Extended access mode conditions.
- File editions—Allow the user to specify file edition numbers or default to the latest edition.
- Accounting and performance statistics—Recording of station accounting and usage statistics.
- Experiment with distributing certain data management functions, which are now performed in STAR, to the stations.

STORAGE STATION MESSAGES

The following list gives messages which can be processed by storage stations. The underlined parameters are returned with the response.

Function	Parameters
	File Messages
Create and open file	*F*, M, M_o, M_p characteristics, name and user ID
Open File	F, M, *characteristics*, name and user ID
Close File	F, characteristics
Close and delete file (temporary and permanent)	F
Close and delete temporary file	F
Keep file	F
Set file characteristics	F, characteristics
Set file length	RB
Is file open	F, *characteristics*, name
Read file pages	F, N, S, B
Write file pages	F, N, S, B
Read file descriptor	F, B

Function	Parameters
Write access list entry	F, M, user access key
Modify owner and public access	F, M, user access key
Mount (tape, pack, cartridge) label L	L
	Test Messages
Read N blocks from storage unit	B, N, S
Write N blocks from storage unit	B, N, S
Read N blocks from storage unit with header	B, N, S, Header
Write N blocks from storage unit with header	B, N, S, Header
Storage unit status	N^1, N^2, N^3, N^4

Legend for Parameters

F = active file index (given by storage station)

M = access mode (used on open)
bit 0 set means cannot delete
bit 1 set means cannot alter access modes
bit 2 set means cannot write
bit 3 set means cannot read

M_o, M_p = access modes of owner and public, respectively (used on creation)

N = number of blocks/file pages to be transferred

S = starting file page number (starts with zero)

B = core block number, if bit 0 set B = SBU address

User ID = user access identifier, variable length string of characters which ends with the record separator character.

N_1 = total number of blocks
N_2 = number of disabled blocks
N_3 = number of active blocks
N_4 = number of free blocks
L = label on pack, cartridge, tape
RB = Length of file in blocks

Message header format

Preceding each set of message parameters is a header which has the following format.

RESPONSE CODE	MESSAGE LENGTH	PRIVATE USE OF SENDER	PRIVATE USE OF SENDER	
PRIVATE USE OF SENDER	TO ZIPCODE	FROM ZIPCODE	MESSAGE FUNCTION CODE	
16	16	16	16	BITS

Details of the message formats are not significant here, except to mention that it is valuable to limit the number of different formats used and to ensure field lengths are large enough to cater for future storage devices. The format is important, however, in respect that once it is established and used by a number of routines even small modifications to it can have widespread effects and are often time consuming and difficult to checkout.

CONCLUSIONS

The storage and file functions of a general-purpose computing system have been identified and separated to operate outside and in parallel with the central processor in a stand-alone, local or remote, storage station. This station forms part of an overall plan to distribute specific functions associated with general-purpose computing into separate computing elements or stations. The same basic hardware and software is used in all these stations to lower manufacturing costs by high volume production. The features and performance of this station have worked out well on delivered and in house systems using drums, large disks and disk packs for archival and working store on both large and small computers. The main reason for success has been the clear identification of the basic file and message functions required and a careful implementation of these functions, utilizing both hardware and software techniques on a standard STAR peripheral station. Although designed to meet the needs of the STAR-100 processing unit the storage station is well suited to be used with any processor which matches its channel and message protocol; it is also relatively independent of storage device type and system configuration.

ACKNOWLEDGMENTS

This work was performed in the Advanced Design Laboratory of Control Data Corporation in St. Paul, Minnesota. The head of this laboratory and chief designer of the CDC STAR-100 and STAR-1B Computer

Systems is J. E. Thornton. The success of the project is mainly due to his leadership and support, together with the hard work over a number of years of the following people in the Advanced Design Laboratory's peripheral group—N. G. Horning, W. C. Hohn, D. J. Humphrey, L. H. Schiebe, E. V. Urness, D. A. Van Hatten, C. L. Berkey, D. C. McCullough and R. A. Sandness.

REFERENCES

1 J E THORNTON
Design of a computer—The Control Data 6600
Scott Foresman 1970

2 T H ELROD
The CDC 7600 and Scope 76
Datamation April 1970 Vol 16 No 4 pp 80-85

3 J E THORNTON
System design and implementation
Proceedings of Third Australian Computer Conference 1966 pp 90-102

4 P D JONES C J PURCELL
Economics and resource parallelism in large scale computing systems
Proceedings of Fourth Australian Computer Conference 1969 pp 241-244

5 P D JONES N R LINCOLN J E THORNTON
Whither computer architecture
Proceedings of IFIP Congress 1971 pp TA4/162-TA4/167

6 W R GRAHAM
The parallel and pipeline computers
Datamation April 1970 Vol 16 No 4 pp 68-71

7 D J WHEELER
Assessing the complexity of computer systems
Proceedings of IFIP Congress 1971 pp I/164-I/168

Protection systems and protection implementations

by R. M. NEEDHAM

University of Cambridge
Cambridge, England

INTRODUCTION

The paper discusses the nature of systems for protection of information in the central memory of a computer, describing the potentialities and limitations of a variety of approaches. It is based upon work done in the course of a current project on protection systems at the Computer Laboratory, Cambridge, and outlines a system which is being developed to the point of hardware implementation in the Laboratory.

PROTECTION SYSTEMS AND PROTECTION IMPLEMENTATIONS

For the purpose of this paper Protection is understood to refer to logical and physical mechanisms for controlling access to data in the central memory of the computer. The purpose of protection systems is to insure that at any point in the execution of a job by means of the computer, only those data objects which require to be accessible are accessible, and that this access is only of the mode, for example reading only permitted, which is required for performance of the task in hand. The object of work on protection systems is to devise mechanisms which will afford protection to the greatest extent possible, and do so without excessive expense in hardware, runtime, or program size. The hope is that if such mechanisms can be devised, then it will be very much easier to contain and to localize the consequences of hardware or software failure, and to know much more precisely than is the case at present which of the activities in which a computer is engaged must be suspected of having been spoiled by the failure, and must therefore be re-initiated.

In order to get any rationale for a protection implementation, we must set up some defined concepts in terms of which protection systems can be discussed. The first of these is that of the segment, the unit of information to which protection applies. A segment is a set of words whose addresses are contiguous in a virtual address space, and whose protection status is at all times the same. Protection is thus intimately bound up with addressing, since our very definition of the unit of protection is in terms of an addressing mechanism. This approach allows us to specify a protection regime by giving a list of those segments accessible to a process at a particular time, together with notes as to the kind of access which is permitted. A somewhat minimal protection regime could then be described by saying that segment A contains data to which read-write access is permitted, while the words of segment B may only be executed as instructions. A major object of research in protection implementations is to propose mechanisms whereby any desired protection regime can be implemented, with as few limitations as possible imposed by the engineering approach adopted.

Protection regimes are not constant during the life of a process. They may change as the work proceeds, and in a fully general discussion they should be allowed to change arbitrarily. Statements would be allowed, for example, to the effect that certain segments were only accessible if the value standing in a system microsecond clock were prime. In practice, one departs from full generality, and limits those circumstances which may give rise to a change of protection regime. A reasonable approximation is to say that changes of protection regime are associated with changes of the segment from which instructions are currently being extracted; this is not to say that such segment changes must necessarily give rise to changes of protection regime, but only that no change of protection regime may occur without a change in the program segment.

The first proposals for the physical design of a processor which took these ideas seriously were by Yngve and Fabry.[2] A summary of their ideas will be found in Wilkes.[1] The essential aspect of these proposals was that there was no restriction on them imposed by any of the implementation techniques. It was thus possible to arrange, in principle, that a process's capability list always contained exactly and only what it should. Yngve and Fabry adopted the same approach to change of protection regimes as we have, namely that it only occurred when there was, additionally, a change of the program segment. A special instruction, called ENTER, caused a complete replacement of the process's capability list, and could thus change the protection regime of the process in an arbitrary way.

In a capability system of the type just described there are two problems calling for further discussion. First, if a capability indicates the absolute store address of the segment to which it refers, there is the problem of updating all copies of a capability when the segment is shifted in memory, and in deleting all copies when, and only when, the segment is destroyed. An obvious solution is to centralize the lists of absolute capabilities, and replace the capability lists associated with running process by lists of pointers to the central list. This is more than a simple technological device because it conceptually replaces the current capability lists of a program by a mechanism which *selects* from a larger list. This selection function has come to seem more and more important to us. Secondly, the original proposals dealt rather clumsily with pieces of data which were the property of a process, in the sense that if the process were deleted the data would go too, but which were only accessible when the right pieces of code were being executed. On the other hand, the proposals dealt very elegantly with bundles of capabilities which invariably became accessible when a certain piece of code was used, regardless of the process using it. The idea that will be developed is that the capability list of a process is to be regarded as that which defines a selection from all the absolute capabilities that exist; at any time in the history of the process some other mechanisms make further selections from the capabilities of the process, the selected capabilities being physically accessible in virtue of the current protection environment. Thus we have the idea of multiple levels of selection.

We may now focus on the implementation of protection as the implementation of selection functions among capabilities, where by a capability we mean that which defines the physical position and size of a segment and the access mode allowed. Immediately there are two ways to proceed, which depend on the extent to which addressing is brought further into the protection implementation. One way is to proceed by means of lock and key systems. A lock and key system is one in which any segment, including here a segment containing capabilities, has associated with it a lock. At any stage in the history of a process there is associated with the process a key. Access to a segment is permitted if, and only if, the current key fits the lock of the segment. A lock and key system tends to separate the notions of addressing and of protection. In such an approach, a process may address any segment whatsoever; only those in which the key fits the lock will do other than give rise to violations. There is no relationship between the mechanisms for addressing a given word and the mechanisms for addressing a given word and the mechanisms for validating access other than that which is implicit in the segment being the unit of protection. Accordingly, it becomes feasible to arrange that a segment has the same name, that is to say it is addressed in the same manner, throughout the lifetime of the process or even to go further and to say that all segments are uniquely identified in the computer. This approach has much merit in that it avoids any renaming problems when communication is involved. Unfortunately, it proves extremely difficult to set up lock and key systems which are of sufficient generality to achieve the desired results. Because of the great potential advantages of lock and key systems, the reasons why this is so merit some examination.

LOCKS AND KEYS

Consider a situation in which all distinct protection regimes which can ever occur are identified by name or number. One could then imagine a lock and key system in which the key consists simply of the name of the current protection regime and the lock associated with the segment consisted of a list of the names or numbers of all protection regimes in which the segment was accessible, together with the nature of the access permitted. It is clear that this places no restrictions at all on the variety of accessibility patterns which can be implemented. It is, however, a very expensive thing to consider doing; there is no convenient limitation which can be set on the length of the lock, and the process of consulting it to see whether a particular key matched would be extremely slow. All practical lock and key systems which have been proposed work by means of some sort of encoding scheme, the purpose of which is to reduce the locks and keys to a fixed and convenient size. Any such encoding scheme regards the lock and key as being bit patterns between which a certain relation is sought. For example the lock and key may be two parts of a single valid message unit in an error correcting code. If we take this as an example, we see that every lock has to have the right relationship to each key which is supposed to fit it. If we now take a particular lock, it is possible in virtue of the structure of the relationship we have constructed to list in principle, all of the keys which will open it; equally every key can be accompanied by a list of those locks which it will open. We can think of listing out the possible locks and drawing lines from each pointing to the appropriate keys, and also putting in lines in the inverse sense from the keys to their locks. We shall be able to express the total variety of protection regimes we are interested in if, and only if, we can make an assignment of locks and keys to the segments in the regimes in such a manner that invalid access is never allowed. This poses an extremely difficult combinatorial problem in all non-trivial cases. It is at the least an extensive task to find allocations which satisfy all the constraints and, even if one can succeed in doing so, a small change in the protection regimes to be implemented may result in a total upset to the lock and key allocation. It appears that one either has to put up with the necessity for computing allocations of locks and keys, or alternatively to accept a lock and key system which will not implement all the protection regimes which might be required. One can sum up by saying that sufficiently powerful lock and key systems are too difficult in practice because of the allocation problem, and that lock and key systems in which one can face the allocation problem are not powerful enough. A good example is the plain hierarchical protection system afforded by representing the locks and keys by small integers, and saying that access is permitted if, for example, the key is less than or equal to, the lock. This is easy to think about and easy to implement; unfortunately, it places extreme restrictions on the protection regimes which can be described. If in protection regime A, something has to be accessible which was not accessible in protection regime B, then necessarily everything which is accessible in B must be accessible in A too. It is just not possible to deal with some situations which occur commonly in practice, such as the following. Suppose there is an input program P_i which has to have access to an input buffer B_i; suppose further that there is an output program P_o which has to have access to an output buffer B_o. It is not possible to arrange that each of these programs has access to its buffer but not to either of the others. These are simple consequences of the linear arrangement of privilege.

An additional difficulty about simple lock and key systems is that they do not deal satisfactorily with the the non-static and unpredictable nature of protection regimes. Arguments which have been passed to a program which runs in a particular protection regime may carry with them the requirement that during running certain segments are accessible because they contain the data passed and not because they are permanently associated with the called program. A simple hierarchic system is in no difficulty if the new protection regime is further up the hierarchy than the previous one, but it is in very serious difficulty if the new protection regime is lower down than the previous one. The more one elaborates lock and key systems, the more this problem becomes a troublesome addition to the allocation problem mentioned before. For these reasons, after a great deal of investigation, we did for the time being abandon the use of lock and key systems as a means of implementing the selection we desire.

SELECTION BY INDIRECTION

As foreshadowed above, the obvious alternative means of selection for accessible segments is by the use of indirection tables. If all segments are accessed via an indirection table or via one of a set of indirection tables, then it is possible to constrain the selection of available segments in quite arbitrary ways by suitable construction, of the indirection tables. A consequence of the use of indirection tables is that addressing has become much more bound up with the protection implementation. This can be seen by looking at the complete specification for getting at a word of core. In order to specify a work, we must give three pieces of information:

1. which indirection table must be used,
2. which entry in that indirection table indicates the required segment,
3. which word in that segment is wanted.

The first two of these will be called the *segment specifier*, and the three collectively an *address*.

The segment specifier of a segment depends on the protection regime, and so in turn does the address of a word. If the protection regime changes, a new set of indirection tables will be brought into use, and the addresses of words will in general change too. What changes is not the segment itself, nor is it the capability for the segment; the change is to the means of finding the capability.

This point is of the greatest importance, and it is worth recapitulating it in a sharp form. On one side of the divide we have systems such as those which rely totally on locks and keys, where if a program attempts to load the accumulator with the contents of a certain word, then the actions it undertakes are in all circumstances the same, regardless of protection regime, although in some protection regimes they may cause a violation. On the other side of the divide, we have systems in which protection is so bound up with addressing that the bit pattern to be presented in order to load a certain word into the accumulator differs according to the current protection regime. The latter approach gives the flexibility which we have been unable to achieve in the former. However, if the mode of addressing words or segments is influenced by the protection environment in force, then there are complications in the compilation process that do not arise in a system with permanent segment addressing. Secondly, one gets into some difficulties with pointers from one segment to another. If we have a data structure which exists in more than one segment, some of the pointers in one segment will point to places in another segment. If the specifier of the segment changes, we are in difficulty. Although this does not happen very often, a solution must be found. The non-uniformity of treatment of pointers is something which compiler writers dislike since the existence of the non-uniformity may not be evident at a convenient time in the compilation process.

Bearing in mind the above points, we now look at methods of implementation of systems which rely upon indirection to perform selection. The principal choice we have considered is between a system with explicitly named capability registers, and one without. A system with explicitly named capability registers works in the following way. A number of registers are provided, usually about eight, each of which is able to contain a capability for a segment in absolute form. Typically this consists of a base, a limit, and an access code. A process is at any time equipped with one or more *capability segments*, which contain either absolute capabilities, or information from which absolute capabilities may be found or constructed. The system has an instruction called 'load capability register' which has two arguments. The first argument is the number of a capability register to be loaded, and the second is an indication of which capability is to be loaded there. It must indicate which capability segment to use if there is more than one, and which entry in the selected capability segment should be used. A store reference instruction will then be interpreted via a capability register. A subsidiary point is whether or not the selection of which capability register to use is part of the address field of the instruction or part of the function field. The significance of this point is whether or not the capability register selection can be changed by index modification. Take first the case where the capability register selection cannot be changed by index modification. In this case a particular instruction in the program has it fixed for all time which register is going to be used. This approach imposes a rather considerable lack of flexibility. Some of this lack of flexibility is associated with any explicit capability register scheme, and will be mentioned in a moment. One aspect, however, is unique to this approach, namely that it is impossible to have a pointer from one segment into another. There is no uniform way of writing a program which will follow a chain searching for something, if that chain is likely to pass through words of more than one segment. It was remarked above that there are difficulties in this area anyway, and possibly the solution to the problem is to decide that intersegment pointers should be disallowed.

Turning now to the alternative case where the capability register selector can be altered by index modification, we see that the particular difficulty just referred to does not arise. Provided that the capabilities for the segments in which the data structure resides are loaded, and known to be loaded, into the correct registers (where 'correct' means the ones which were assumed when the points were set up), then inter-segment pointers are perfectly possible. This proviso, however, indicates the lack of flexibility which remains. A great deal of pre-allocation of capability registers has to be done in any system which refers to them explicitly. Furthermore, an instruction will only be correctly executed if the right capability register has been loaded. Unless there are sufficient capability registers, which may be rather a lot, there is a good deal of keeping track to be done to insure that at all times the correct capabilities are where they should be as the flow of control proceeds round the program. For example, it may be desirable to pass the address of a word around in a program at a time when a capability for the segment containing it is not necessarily loaded. There is of course no need for the capability to be loaded until the address is actually used. We find that we need, in effect, two sorts of address which can be described as a *particular* address and a *general* address. A particular address consists of a capability register number and an offset. It is valid in all circumstances in which the capability register has been properly loaded. A general address consists of a complete segment specifier and offset; the segment specifier is just the second argument of a 'load capability register' instruction. If a piece of program, say a sub-routine, receives a general address, it is in a position to load the indicated capability into whichever capability register it thinks fit. However, in this case also we have difficulties of compilation, because the compiler cannot know when to use general addresses and when to use particular addresses. Furthermore, considerations of economy would suggest that we do not need two forms of address; of the two it is clear that the general one should be retained.

THE SYSTEM PROPOSED

This final remark leads to the outline of the system we have eventually proposed. There are no explicitly named and explicitly loaded capability registers; instead the general address as defined in the last paragraph is interpreted directly by the hardware. The hardware

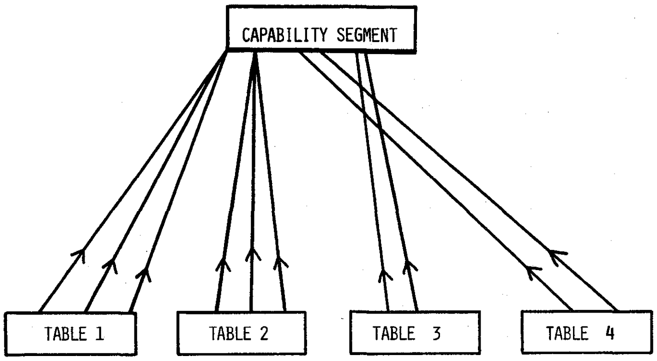


Figure 1

must internally have registers in which absolute capabilities are to be found, and what it does, when presented with a general address, is to test whether the absolute capability corresponding to the segment specifier part of the general address has already been loaded into one of the internal registers. There are a variety of ways of doing this at hardware level. We are now in a position where programs only use addresses in the form 'segment specifier, offset', and the runtime interpretation of the segment specifier is buried beneath the hardware-software interface. We must remember, however, that the interpretation of a segment specifier will still depend on the protection regime, because it makes use of indirection tables as a means of selection.

It is now time to return to a question implied about, namely how many indirection tables there should be and what they should be used for. The structure we are talking about is sketched in outline in Figure 1.

In this structure, a change of protection regime will be implemented either by changing the contents of the indirection tables, or by bringing into use new indirection tables and putting out of use old ones. Some things are most naturally done by amending the contents of indirection tables. For example, a system call to give the process a brand new segment results in a change to the protection environment which is most easily made by extending a presently existing indirection table. The call has said something like 'get me a new segment of size n and call it Jack' where Jack is a segment specifier. The consequence will be that the appropriate indirection table entry will be set. On the other hand, when protection regimes change not by giving the process new resources but by changing the accessibility of the resources already given, it is expedient to bring new (but pre-existent) tables into use and similarly to dispose (temporarily) of old ones. We have chosen to classify the segments available for a process at any time into four classes, implying that there are four current indirection tables:

1. Segments which are available to the process regardless of which program is currently being executed; these are known as G for global.
2. Segments which contain the code, or alternatively read only data, for a current program; these are called P for program.
3. Segments which, although the property of the process, are only accessible within the current program; these are called type CP.
4. Segments which are accessible because they have been passed to the current program from the program which called it; these are called type A for argument.

For example, consider a program package whose duty it is to perform an input/output operation, such as taking a string of characters away from the calling program, despositing it in a buffer, and subsequently disposing of it. The code of the package may read from or write to the calling program's data area, it will require to be passed capabilities which will be A type. If in the course of executing this package it is necessary to make calls to the generally available operating system facilities, the ENTER capabilities for these facilities will probably be capabilities of type G. If, however, the system calls may only be made from within the input/output package we are describing, those ENTER capabilities could be either of P type or of CP type. The action of an ENTER instruction will thus be to change three of the four indirection tables. The table G will not be changed, because it is always available. The P indirection table will be replaced by one which is the defining characteristic of the called package; everything referred to in the P table will be shareable between all users of this procedure. The existing CP table will be replaced on ENTER by one set up to have the required properties at the time when the procedure was made available to the process. Making a procedure available to the process thus consists of equipping the process with the required ENTER capability and with the required indirection tables. The A indirection table will be replaced by one which is characteristic of this particular call. It is convenient to place A indirection tables on a special stack of standard format and distinguished from any stack that the running program may create for its own purposes. The special stack can also be used to store the links associated with ENTER instructions. Specifically, if an A type indirection table is constructed before a call, it will be the top few words of the stack. One or two special instructions are provided for moving pointers to capabilities from one indirection table to another, and one of these is specifically used for establishing entries in what will be a new argument type indirection table. It is worth noting that in the system proposed material other than that in global segments will only be available to called programs if appropriate capabilities are explicitly passed. There is inevitably a slight overhead on calls, but this is unavoidable in any system which does not have hierarchical protection. In hierarchical systems, it is usually assumed that when a call is made to a more privileged regime (and most calls are like this) everything which was previously available is still available.

We are now in a position to give some account of the protection system as it appears when a process is running without any reference to problems of interprocess communication or of coordination. At any time the protection regime is represented by the current settings of the four indirection tables. Some of the capabilities referred to in these indirection tables will be ENTER capabilities; these delineate those changes of protection regime which are immediately possible. When one of the ENTER capabilities is exercised by means of the ENTER instruction, the protection regime changes and the P, CP, A indirection tables are all replaced. We thus see that an ENTER capability must specify, directly or indirectly, the capabilities for the two new indirection tables of the P and CP types, the A type being part of a stack as previously described. What an ENTER capability actually looks like in a process capability segment is an implementation decision.

We can now look at the same questions from another angle, and consider how to construct a protected procedure—that is, a procedure which will be entered with an ENTER instruction and which will run in its own protection regime.

A protected procedure is characterized by its P- and CP- indirection tables. Accordingly, to construct one we must construct these tables, and insure that there are in the process's capability segment the correct capabilities for the indirection tables to select. A specimen prescription for such a procedure could look something like this:

"There are 4 entries in the P-table. The first must select a segment of program whose text-name is Peter and the only access needed is 'execute.' The second selects a translation table called Bill, and 'read' access is required. The third and fourth must select ENTER capabilities for two standard system functions.

"There are 2 entries in the CP-table. One is for local workspace of the procedure, and should be a copy of named segment Alfred, which contains initial data values. It must be readable and writeable. The second must be a workspace segment to use as a buffer, readable and writable, and 1000 words long."

A routine that interprets this prescription and sets up an ENTER capability for the procedure in question then takes the following actions. First it procures suitable segments in which to build the indirection tables, and then it sets about filling them in. In the case of a workspace segment, whose initial contents do not matter, all that is necessary is to ask the core management routine for a segment of a suitable size and set the appropriate pointer in the indirection table. In the case of a segment whose initial contents must be set from a file, then the file system must be consulted in order to discover the segment size and disc address. There is a third possibility, namely that the prescription is for a segment already known to the process, and in this case the insertion of a new pointer is all that is needed. The two entries for standard system functions mentioned above would very likely fall under this case. Since the purpose of the routine is to equip a process with a new ENTER capability, it may be convenient to write it so that it can act recursively when the prescription itself calls for ENTER capabilities. The final action of the routine is to construct the ENTER capability which was originally requested, and leave a pointer to it in a suitable place.

In this approach the protection procedure is regarded as a totally encapsulated entity which can be incorporated into the environment of a process without any presuppositions as to what was there already. If parts of the (read- or execute-only) environment were present already, then they will be re-used. It is open to take a slightly different approach and to construct protected procedures on the assumption that, for all processes, certain standard functions are available through the G-indirection table, this being always accessible. Doing this makes P-indirection tables shorter, but requires more conventions as to the way processes are set up.

THE PROBLEM OF INVALID ARGUMENTS

It is common for one or more of the arguments of a call to a protected procedure (or indeed any procedure) to be the address of a piece of store to which the procedure will write. There is no protection problem if the store so addressed is accessible to the calling program; potential difficulties arise if it is not so accessible but would be accessible to the called program. As a concrete example, suppose that in a traditional computer where the supervisor runs in a privileged mode, all memory being accessible, there is a system call to read n words from an input document to store starting at address a.

If a user program executes this call, giving as argument an address in store available to itself, there is no problem; what, however, if the address is that of store inaccessible to the user, but accessible to the supervisor? Unless precautions are taken, the supervisor may, when presented with an invalid argument, over-write its own program or important data. This problem is not new; there are explicit counter-measures to it in the hardware of the Atlas. However, the more generalized one's approach, the more difficult it is likely to be to deal with this class of difficulty.

In a system with explicitly named capability registers, and in which the capability register number is in the function part of an instruction (i.e., it cannot be altered by address modification) the problem cannot arise. This is because any address passed as an argument will only be interpreted by the called program as referring to an authorized segment, and no possible action can mislead it. As soon as we move to a system in which the capability register number or the segment specifier are parts of the address passed, then there is the possibility of trouble. Difficulties of this sort arise in any system in which indirect references to segment names or numbers are possible.

In order to guard against the danger just referred to a check must be made which depends on a number of different pieces of information being available at the same time. We must know:

1. The protection regime in which the address was constructed;
2. The protection regime in which the word referred to by the address is accessible;
3. Whether it is allowable to construct an address in the former regime which refers to a word in the latter.

In the structure outlined above, where there are four indirection tables, a simple rule results as follows: an address residing in a segment of type G or A may not specify a word in a segment of type P or I. The difficulty comes in knowing when the rule is being broken. As an example of a common sequence in which the relevant information is not all at hand at once, consider:

Load index register from store
Access store via index register

Item 1 above is available in the first instruction, but it is not then known that the word will be used as an address. Item 2 is known on the second instruction, but not where the contents of the index register came from. Any approach, for example the use of indirect instructions, which has both pieces of information available at the same time will enable the problem to be solved, e.g.,

'Load accumulator indirectly from store'

because *both* addresses and hence both segment types are know in the course of the same instruction, or

'Validate stored address'

which is like 'Load accumulator indirectly' except that it does not load the accumulator, but only checks that the address in store obeys the rules. A really satisfactory solution to the problem of invalid argument addresses would not place on programming style the constraints which are imposed by the compulsory use of indirect or validation instructions. Such a solution is not yet obviously available.

The body of this paper has been concerned with protection systems within a process. Nothing has been said about how the process obtains its resources and from where. There follows a brief view on how this aspect of a system may be organized.

The time available to a process is administered by a superior process called its coordinator. The coordinator is responsible for allocating time to its junior processes, and for synchronizing their execution where necessary by managing their halting and freeing. In addition to being the source of time allocation, the coordinator has responsibility for space allocation. Finally, any process may act as coordinator for processes junior to itself.

This view has consequences for protection. The within-process protection architecture discussed above aids the orderly use of the process's resources, and all privileges conferred on particular procedures are relative privileges within the general facilities available to a process. Since all facilities available to a process are mediated by the coordinator, the last statement implies that privileges are valid within the universe set up by a coordinator for its junior processess, this universe being a subset of that available to the coordinator itself.

It is a consequence of these remarks that privileges enjoyed by a coordinator in virtue of its relationship to its superior may not be passed on to the coordinator's juniors. They exist in the wrong world.

The result then is that a coordinator may pass to its junior processes, when setting them up or later, access to core segments or subsegments available to it, with or without further access restrictions. It may not pass an 'ENTER' capability at all, though it may be able to pass the use of pieces of code from which an ENTER capability can be constructed for the junior. Since the coordinator has complete control over the actions of its junior processes, including interfering with register settings during halts or after interrupts, passing an ENTER capability could allow the coordinator to perform, via a subordinate, action which would ordinarily be forbidden. The ENTER refers not merely to a piece of code but to package whose existence implements privileges granted by the coordinator's superior.

In the above approach, there is nothing unique about the status of a coordinator. Any program may create subprocesses for which it carries out coordinator functions according to any queueing logic or discipline it may choose. Two instructions are to be provided in our experimental system to assist in this operation; 'ENTER SUBPROCESS' which effects the complete change of protection context required by making current a new process capability segment and new indirection tables—the new capability segment being defined by reference to the old—and 'ENTER COORDINATOR' which reverses this action.

CONCLUSION

The foregoing discussion has attempted to describe the requirements upon a protection system for information in central memory, and to bring out the problems which arise from various approaches. The upshot is an abandonment for the most general protection systems of lock-and-key methods, and the use instead of methods which rely on selection by indirection. It should not be forgotten, however, that if the requirements of a protection system are modest, then a lock-and-key method may well be feasible. An outline was given of a practicable indirection technique for use in more general cases; again it should not be forgotten that others can be devised which may be more suitable in particular cases.

ACKNOWLEDGMENTS

I am much indebted to Professor M. V. Wilkes who has devoted a great deal of effort to improving the text of this paper, as well as many discussions of its content. The ideas have benefited from interaction with numerous colleagues, in particular three research students, J. R. Horton, C. C. Kirkham, and R. D. H. Walker. The work is part of a project supported by the Science Research Council.

REFERENCES

1 M V WILKES
Time sharing computer systems
Second Edition American Elsevier 1972

2 R S FABRY
Preliminary description of a supervisor organised around capabilities
Quart Prog Report No 18 Section IIA Inst Comp Res Univ Chicago 1968

Burroughs B1700 memory utilization

by W. T. WILNER

Burroughs Corporation
Goleta, California

INTRODUCTION

Squeezing more information into memory is a familiar problem to everyone who has written a program which was too large to fit into memory. Program compaction is also important to those who work on machines with virtual memory (such as the B5500[1]); despite the almost unlimited amount of storage, one wants to keep program working-sets[2] (collections of segments needed in core at the same time) as small as possible to reduce both the number and duration of segment swaps. In general, one seeks to raise the information content (or reduce the redundancy) of the blocks of information which one is using. In this discussion, "information content" will suffice as an intuitive notion.

One of the devices which hardware and software designers have provided to help with compaction is choice of container sizes. Machines can manipulate more than words: bytes, double words, and so on. Languages allow variables to be declared with different sizes, e.g., four-byte or eight-byte integers. Another category of compaction devices is encoding techniques. For example, memory addresses may be encoded literally, or as a "base-register-name/displacement" pair, or as an "indirect-reference-flag/reference-table-index" pair, or so on. A third technique for raising information content is to group information according to time, that is, by keeping information which is likely to be needed at the same time in one place. For example, variable-length segments are more efficient than fixed-length pages,[3] partly because segments are made to contain coherent subprograms, which is a way of grouping according to time.

Ideally, then, a computer system very likely to utilize memory most efficiently would be one which could (a) manipulate any size bit string, (b) interpret any sort of encoding, and (c) administrate any segmentation scheme.

UNIQUE DESIGN REQUIREMENTS

Burroughs B1700 (described elsewhere in these Proceedings[4]) is the only information-processing system (known to the author) which almost attains these ideals. The B1700 is specifically designed (a) to manipulate fields from zero to 65,535 bits long equally adeptly (which is a requirement of its *defined-field design*), (b) to interpret arbitrary "soft" machine language, or *S-language*, faster than a hard-wired system in the same price class could execute identical functions (which is a requirement of its generalized language interpretation design), and (c) to automatically move information in and out of memory according to any scheme (which is a requirement of its throughput objectives). As a result, the information content of fields in B1700 memory is exceptionally high, and memory is often utilized twice as efficiently as on other systems.

COMPACTION TECHNIQUES

Arbitrary field size

With defined-field design, fields may be defined to be just the size that is necessary, however many bits that may be, and other, arbitrarily-defined fields may begin in the very next bit. One bit will do for boolean variables, and it may truly be any bit in memory. Character strings may begin on any bit address. There is no such thing as byte alignment, or data specification. A major addressing boundary, if it can be called that, occurs between each of 2^{44} (over 17 trillion) bits. Every bit can be fully utilized.

There are no locations and no field lengths which offer any processing advantage over other locations and lengths. Therefore, S-language designers are free to choose container sizes, such as for data addresses, which are precisely as many bits long as desired. This simple freedom appears to account for half of all the program compaction which has been realized on the B1700.

S-language designers are further able to leave such things as branch address field lengths unbound until after compilation, when specific program details are known, such as the maximum number of instructions to be skipped by a branch instruction. It is just as easy to bind field lengths at run time as earlier; hence, S-language format can profitably change from program to program.

Frequency-based encoding

Given that fields may have arbitrary sizes, S-language designers (and users) may employ the varying-size containers generated by Huffman's algorithm for minimum redundancy codes.[5] Briefly, the technique encodes elements by means of strings whose length varies inversely with the occurrence frequency of the elements; i.e., the most frequent element is represented by one of the shortest strings, and the least frequent element is represented by a longest string.

Huffman encoding constitutes one extreme form of representation, which may possibly stipulate a different length string for each element to be represented. The opposite extreme is uniform container size, e.g., words. Between these two extremes lie a range of encodings, which particular circumstances may merit, as will be illustrated later.

As a simple illustration of frequency-based encoding, suppose a defined-field computer with a six-instruction repertoire exhibited the following frequency counts of instructions in a program whose size was to be compacted:

Instruction	Frequency
#1	1000
#2-#6	5@200
Total	2000

Using ordinary encoding techniques, a three-bit field would be used to represent six quantities. The program's 2000 instructions would then be represented by 6000 bits. If, on the other hand, we allow variable-length, frequency-based encoding, the most frequent instruction could be encoded with only one bit. The bit would signify either the instruction or that three more bits follow, carrying the encodings of the remaining five instructions, viz.:

Opcode	Instruction	Occurrence	Total Bits
1	#1	1000	1000
0111	#2	200	800
0110	#3	200	800
0101	#4	200	800
0100	#5	200	800
0011	#6	200	800
			5000

One thousand bits are eliminated, increasing memory utilization by:

$$\frac{6000 - 5000}{6000} \times 100\% = 16.7\% \qquad (1)$$

A better encoding would use two bits for one of the five less frequent instructions, since the remaining four could still be encoded in four bit opcodes, viz.:

Opcode	Instruction	Occurrence	Total Bits
1	#1	1000	1000
01	#2	200	400
0000-0011	#3-#6	@ 200	3200
			4600

Fourteen hundred bits are eliminated, increasing memory utilization by 23.3 percent. Note that this encoding has no unused bit combinations; it can be used for exactly six instructions. More redundant codes have room for other opcodes.

Time-based representation

In addition to representing information in fields according to occurrence frequency, one may improve memory utilization by rearranging fields according to dynamic frequency. That is, fields which are needed most often in memory may be collected into a common segment, in a time-analogy to minimum spatial redundancy. The B1700's interpreters are equipped to record program profile statistics[6] which determine what pieces of code spend the most time being executed. By designing S-languages which allow arbitrary grouping of data or program pieces into segments, one may permit program representations in which most-often-used constructs appear in short, coherent segments while relatively unused portions reside in large, discontinuous (from the standpoint of flow of control) segments. Bits in each segment have similar time-utilization, just as the varying length of fields in Huffman encoding grant similar space-utilization to the bits in a particular field.

Dynamic field size

A defined-field computer must transmit a field length as well as a bit location to memory for each access since arbitrary field lengths are permitted. Consequently, it is just as convenient to have operand lengths dynamically changeable as fixed. Length constants must be stored somewhere between requests to memory, and it is no less efficient to keep them in addressable fields. This opens up the possibility of Dial-A-Precision FORTRAN, where the operand fields in the FORTRAN S-language can be adjusted on the fly to be long enough to hold a required precision, for example, during inner product calculations. This capability is planned for the B1700 software, but is not in the initial releases.

APPLICATION TO SPECIFIC S-LANGUAGES

Since all high-level languages on the B1700 are compiled into novel S-languages of Burroughs' own invention, opportunities exist in these contexts for improved memory utilization. S-languages for existing machines, such as System/360 machine language, prohibit compaction because the fields are locked on to non-defined-field hardware formats.

SDL S-language

Burroughs supplies B1700 customers with a language and interpreter which have been designed to be most efficient in a compile-time environment. Named Systems Development Language, SDL, it has been used to program all B1700 compilers. SDL is constructed from an extendable base language which has been used, in augmented form, to write the B1700's Master Control Program (which performs supervisory functions such as I/O, multiprogramming, multiprocessing, virtual memory management, etc.) and, in a different form, to write sorting applications.

SDL opcodes

Opcodes are encoded into three lengths: four, six, and ten bits. Of the sixteen four-bit combinations, ten name the most frequent instructions, five indicate that two more bits specify the remainder of a six-bit instruction field, and one signifies that six more bits are needed to define the operation. The design trade-off between space and time in opcode representations does not vary linearly between the extremes of Huffman encoding and fixed container size. One fixed field length allows parallel decoding of all bits in the field, minimizing time, but requiring much storage (except when all elements have identical occurrence frequencies, but that is contrary to computer behavior). Huffman codes may require much more decoding time, since bits may need to be examined serially until the length of the field manifests itself, but the codes can minimize storage. In the middle, SDL's three lengths come very close to minimizing storage, and also incur very little extra decoding time, as Table I indicates.

TABLE I—Comparison of SDL Opcode Encoding Against Extreme Methods

Encoding Method	Total Bits for MCP's Opcodes	Utilization Improvement	Decoding Penalty	Redundancy
Huffman	172,346	43%	17.2%	.0059
SDL 4-6-10	184,966	39%	2.6%	.0196
8-bit field	301,248	0%	0. %	.4313

Figure 1 presents the same figures graphically. The reason for Figure 1's exponential curve is that there are several orders of magnitude between the frequencies of the most and least frequent elements in the set to be encoded. There is a great deal to be gained in such circumstances even by encoding the single most frequent element in a shorter field than the others (as was illustrated also in our example). If the opposite were true, if all elements were uniformly frequent, then the trade-off curve would be linear (or nearly so, depending on what multiple of the encoding radix the number of elements is).

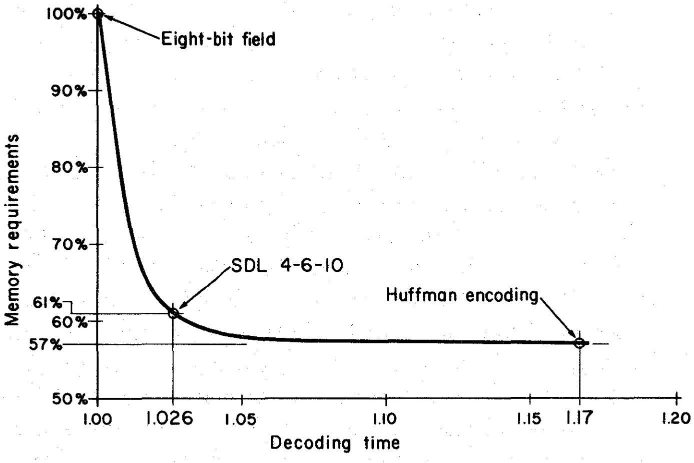


Figure 1—Performance of SDL encoding compared to extreme techniques

Redundancy

We can compare these techniques on a less intuitive basis. "Information content" may be precisely defined in terms of the probability of a message's occurrence (as opposed to its meaning). Shannon's entropy function[7]

$$H = - \sum_{i}^{I} p_i \log p_i \qquad (2)$$

gives a measure of the average information content of I independent events with individual probabilities $\{p_i\}$. If we consider an SDL opcode in the MCP program as an event, and calculate $p_i \log_2 p_i$ for all 73 opcodes, then we find $H = 4.55$, which may be interpreted as the average number of bits needed for an opcode. To compare the encoding techniques of Table I using this criterion, we have:

Average content (bits/opcode)	Technique
4.58	binary Huffman
4.88	SDL 4-6-10
6.51	8-bit field

which shows that our chosen technique is very close to the minimum value of 4.55.

The redundancy factor of an encoding technique may be calculated as

$$\text{Redundancy} = 1 - \frac{\text{optimum message length}}{\text{encoded message length}}, \qquad (3)$$

which ranges from zero (no redundancy) to one (infinite redundancy). To derive the redundancy column in Table I, we compared the total bits in the MCP via each technique against 4.55 × 37,656 (the total number of MCP opcodes) = 171,349, which is the smallest number of bits that may be used under the assumption that opcodes are decoded independently.

Redundancy, despite its quantifiability, is not a good independent design criterion. If pursued too extremely, there are disadvantages (such as intricate and slow decoders). If ignored, of course, there are extreme disadvantages (total system inefficiency). One must consider it in balance with all other design criteria, and attempt to reduce it without sacrificing performance in other areas. Most importantly, one must not sacrifice the unquantifiable criteria, such as ease of use, which appear to be most significant. It is interesting, however, that the B1700's S-language concept (which was pursued primarily to improve ease of use) has the desirable side effect of taking actual opcode representations out of the programmer's attention (because the possibility of working with machine language is removed), and this allows further efforts to remove redundancy, because opcodes no longer have to be human-engineered. Ease of use and high memory utilization are not orthogonal design criteria and increasing one need not decrease the other.

Significance of opcode compaction

Opcodes, in SDL's case, occupy nearly one-third of the entire program space because the choice of S-language significantly reduced all other kinds of fields. Compaction of opcodes contributed the most toward reducing overall SDL program size.

SDL data addresses

Locations of variables, or data addresses, are the second most populous fields, after instructions. SDL is a block-structured language and the SDL machine (for which the SDL S-language is the machine language) is a stack-structured processor, so data is accessed by a pair of integers, one giving the lexicographical level on which the variable was declared in the SDL program, and the other giving an occurrence number, or ordinal position, of the declared identifier in its block. The level identifies a (dynamically varying) region of the stack and the occurrence number indicates a displacement into the region where the variable may be found.

In order to accommodate extremely large programs, the language designers decided to allow up to 1024 variables on any lexicographical level, and up to sixteen nested levels. The largest data address, thus, would require fourteen bits, ten for displacement and four for level. Once the compilers and the MCP were written and debugged in SDL, the actual usage of bits in data address containers was studied, in order to apply frequency-based encoding techniques. Table II gives the usage statistics for the B1700 MCP. Using the arbitrary fourteen-bit container, 9174 addresses require 128,436 bits. The usage study found that 66.1 percent of the occurrence numbers could be contained by a five-bit field and 78.4 percent of the level numbers were either the current level or level zero, which could be encoded in one bit. All together, if these shorter fields were made available, only 94,900 bits would be required, which is a 26.1 percent improvement in memory utilization. By mutual consent of the two SDL compiler writers and the SDL interpreter writer, it was agreed that the S-language would be changed to include a new data address format: level fields of one or four bits, occurrence

TABLE II—Occurrence of Actual Field Lengths Required by B1700 MCP Data Addresses

Level Field Size					Displacement field size						
	0	1	2	3	4	5	6	7	8	9	Total
0	136	31	12	29	67	130	189	345	635	0	1574
1	478	355	397	628	762	1182	1116	701	0	0	5619
2	162	143	272	283	223	408	107	13	0	0	1611
3	56	45	68	86	64	44	7	0	0	0	370
4	0	0	0	0	0	0	0	0	0	0	0
	832	574	749	1026	1116	1764	1419	1059	635	0	9174

Relative Level Contents	0	1	2	3	4	5	6	7	8	9	Total
0	521	416	411	501	409	342	231	108	46	0	2985
−1	234	135	178	354	567	871	689	439	205	0	3672
−2	44	23	159	136	102	454	329	167	300	0	1714
−3	13	0	1	35	29	93	146	305	41	0	663
−4	16	0	0	0	9	3	24	40	41	0	133
−5	4	0	0	0	0	1	0	0	2	0	7
−6 to −15											0

number fields of five or ten bits, and two prefix bits to indicate which of the four possibilities followed. Locations in SDL are thus eight, 11, 13, or 16 bits long.

Because this scheme is so different from conventional techniques, it is difficult to establish the exact advantage in memory utilization. If we consider a conventional scheme which can address as many variables, it is reasonable to require that two bytes of address field be used, since it is certainly possible for a program with 2^{16} variables to be executed by the SDL interpreter. Another way of reaching the same conclusion is to consider the fourteen-bit maximum container; without defined-field design, fields must be byte-multiples (at least), so two bytes are needed. For 9,174 addresses of 16 bits each, 146,784 bits are needed. Hence, the four-way SDL encoding offers 35.4 percent memory utilization.

SDL code addresses

Program points are addressed by a pair of integers‘ one giving a segment name and the other specifying the starting bit of an instruction in the segment, relative to the start of the segment. Program segments are stored separately from data segments.* As a consequence, code addresses may be structured differently from data addresses. This freedom is advantageous for compaction, too, because usage information may be applied independently to each kind of field. Code address requirements are typically very different from data address requirements. Programs usually have many more variables than segments, so fewer bits are needed for segment names than for variable addresses. Segments usually contain more bits (thousands) than blocks contain variables (less than a hundred), so more bits are needed for displacement fields than for occurrence number fields.

SDL designers wanted to allow over a billion bits for programs, in up to 1024 segments of up to one million bits each. At the same time, they surmised that many references to the first 32 segments might better be encoded in five-bit segment names, and references to the first 4096 bits and the first 65,536 bits might be more efficiently encoded in 12- and 16-bit fields, respectively. These shorter options were included in the preliminary S-language design which was used during MCP and compiler construction and check-out.

Prior to release, actual usage was studied to evaluate the appropriateness of the design choices.* A sample of

* This is so that protection can be efficiently implemented and so that reentrancy is free, i.e., more than one program can execute a segment concurrently without requiring a different representation from that which a one-program version would use and without executing any instructions specifically to administrate reentrancy.

* On the B1700, S-language design may be changed at any time. Programmers see only higher-level language which is independent of S-language format. Hardware sees only microcode, which is indifferent to S-language format. Many S-language revisions can, in fact, be implemented simply by changing some literal fields in the interpreter and compiler.

TABLE III—Occurrence of Actual Field Lengths Required for B1700 MCP Code Addresses

Segment	Displacement field size														
Field Size	0–2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
0	4	0	0	0	0	0	0	0	0	153	1	39	33	102	332
1	0	0	0	0	0	0	0	0	0	0	1	0	5	1	7
2	14	0	0	0	0	0	2	4	0	2	1	0	0	0	28
3	0	0	0	0	6	0	0	7	5	9	6	9	0	0	42
4	7	0	0	0	0	0	0	16	42	12	42	28	0	0	147
5	4	0	0	0	0	3	140	19	32	36	66	25	5	0	309
6	4	2	0	2	1	4	13	47	110	37	80	166	36	0	502
7	5	0	0	0	2	18	5	39	78	132	147	168	9	0	603
8	6	0	0	0	0	4	4	23	12	5	40	26	0	0	120
	44	2	0	2	9	29	164	155	268	376	389	461	88	103	2090

2,090 code addresses from the B1700 MCP corroborated the design team's choices, including that of a null segment field, for references within the same segment. Table III gives the occurrence of the actual field requirements for the code address sample. Actually, a fraction of 1 percent improvement in memory utilization could be achieved by changing from a five-bit segment field to a six-bit field, but if future SDL programs turn out to be smaller than present Burroughs compilers, the usage statistics will eventually prefer the present fields, so no change was made. SDL code addresses, then, have the following format:

field description	segment name	displacement
3 bits	0, 5 or 10 bits	0, 12, 16 or 20 bits

Of the eight permitted variations, the most frequent is 15 bits long; three description bits, no segment field, and 12 displacement bits. Except for the null address (only three bits long), this is the shortest option, so it is to be expected that a proper frequency-based encoding of code addresses would make this the most commonly used format.

Comparing this scheme against conventional hardware is no less puzzling than it was for data addresses. Perhaps we may consider that the SDL machine can directly address 2^{30} (over one billion) bits of instruction storage. An equally capable byte-oriented machine needs to address 2^{27} bytes directly, so four bytes are needed. The SDL representation uses 74,303 bits for 3,767 addresses in the B1700 MCP, whereas the byte-oriented machine would need 120,544 bits (ignoring the extra capacity that four bytes allows). In terms of memory utilization, the ability to define eight different code address formats yields a 38.4 percent advantage for the B1700.

SDL profile statistics

While an SDL program is running, the SDL interpreter records each segment transition in an array which is automatically allocated to each program. This monitoring, performed by microcode, adds less than ¼ percent to the running time. At the end of the run, SDL code is interpreted which prints the number of times each segment was entered, a compiler-estimated traversal time for each segment, and the product of the two, giving an indication of which segments used the most execution time. SDL programmers can also indicate at compile-time which sections of their program should be monitored closely, down to each conditional expression, so that exact frequency counts can be obtained for each bit of code in the program. Subsequent monitoring involves additional code, but the amount is always less than a 1 percent increase. Assuming segment and procedure frequency counts have been utilized to focus one's attention on only a few procedures, execution overhead for conditional expression monitoring is under ½ percent in most cases. Thus, the B1700 can indicate to programmers what the time-utilization of their program sections is, at very low cost. By grouping similarly utilized sections into segments so that segment transitions are lessened, temporal memory utilization can be increased.

Other S-languages

Only SDL designers were able to apply accurate usage statistics to their design because the entire world's supply of SDL programs was available to them. COBOL, FORTRAN, BASIC, and RPG (the initial set of languages for the B1700) S-language designers collectively extracted usage statistics from over nine million bits of sample programs, but can only guess at their verity.

COBOL S-language

Opcodes are represented by three- or nine-bit fields; the seven most frequent instructions are encoded into seven of the eight three-bit codes, and the eighth is an escape code which indicates that six more instruction bits follow. Operands are prefixed by a single bit which indicates literal operands or variables. Literals consist of a type field (two bits), a length field (three or eleven bits), and the literal string itself. Operands are indicated by a string of fields which give segment and displacement location, length, type, character code semantics, and subscript information. Frequent operand descriptions do not appear in-line; they are placed into a table by the compiler and their table index is used in-line. COBOL S-language even includes a sub-S-language which defines editing operations. This implies that the COBOL interpreter contains a sub-interpreter which handles the editing program strings. These program strings, like operand descriptions, may appear either in-line or tabulated.

COBOL, more than any other language on the B1700, has taken advantage of the ability to leave object code format undefined until after the compiler has seen a program. Segment descriptions, displacements within segments, variable-length data's size fields, operand descriptors, data addresses, and branch addresses are all stored in fields whose size is made just large enough to hold whatever maximum value is needed for the particular program. This capability appears to reduce COBOL object program sizes by 46 percent; that is, if the length of these containers were fixed for all programs, the average program size would be 85 percent larger. Of course, the amount of compaction is so large because the chosen S-language provides many opportunities to eliminate wasted space.

The overall appropriateness for COBOL's S-language is difficult to assess. One faces the same evaluation problem as trying to say how much better one machine design is than another at implementing COBOL. So far, the only secure comparisons which we have obtained pertain to overall throughput and resource requirements. From a set of twenty ANSI COBOL programs of diverse application and varying size, we have concluded that COBOL programs tend to occupy 70 percent less memory on the B1700 than they do on a System/360 model 30. Such a drastic reduction in memory also improves running speed, which averages around 60 percent faster than the 360/30. The B1700, when interpreting its COBOL S-machine, even seems to out-do the B3500 system, whose hardware was designed to execute and compile COBOL programs. Program storage requirements are 60 percent less, and execution times are comparable.

RPG S-language

In order to reduce the number of interpreters active at any one time on single-processor B1700s, the initial release of RPG uses the same S-language as COBOL. From a set of 31 RPG programs used for benchmarks, we observed that program storage is typically 50 percent of System/3 size (although one program with a preponderance of character strings in its representation was only 25 percent smaller, due to the fact that eight bits are used on both systems to represent characters). Execution speed is between 25 percent and 50 percent faster than System/3, due to the conciseness of the instruction stream and S-language advantage. COBOL S-instructions are interpreted at an average rate which is six times slower than System/3's average instruction rate (36 usec. vs. 6 usec.). To achieve 50 percent faster running time, each S-instruction must, on the average, accomplish twelve times more work than each System/3 machine instruction. Obviously, size alone cannot adequately measure program compaction.

FORTRAN S-language

FORTRAN also uses an opcode format of three or nine bits in each field, with seven short ops and one escape code. Data and code addresses have a common format, usually 24 bits long: field description bits (six) which control interpretation of the rest of the address and of the operand as well; a segment field (ten bits) which either names a segment or a place where a segment name can be found; a displacement field (eight bits) which locates operands within a segment; and possibly more fields, depending on context.

Summarizing seven moderately-sized jobs, FORTRAN programs tend to occupy 50 percent of the space needed on a System/360, and 40 percent of the space needed on a B3500. (Note that these figures and those for COBOL imply that the B3500 is better at representing COBOL programs than a 360 and not as good at representing FORTRAN programs, which is well-known.)

DISCUSSION

Inherent limitations

Although choice of S-language format has been completely free on the B1700, there are implied restrictions due to the semantics of the higher-level languages for which S-languages have been invented. Contemporary languages, and FORTRAN especially, reflect the kind of hardware which their designers knew existed: sequential, word-oriented processors. Defined-field design offers significantly different machinery, and languages have yet to be defined which unconsciously assume defined-field capabilities, namely, that data and programs may be represented in any format whatever. Good as the B1700's memory utilization is, it tends to be better for programming languages which have a large number and variety of data and program formats.

Diminishing returns

Modifying interpreters to accommodate S-language refinements has a definite cost, including reprogramming affected portions of compilers and recompiling source programs. Our experience indicates that after identifying and improving seven or eight redundant aspects of a language, information content is relatively uniform among various S-language fields. Further refinements may not be worthwhile. This also implies that only first-order usage statistics need be collected, which keeps analysis costs down.

User optimization

When several alternative encodings seem equally attractive and their design trade-offs are well drawn, their invocation may be placed under user control. Each programmer knows individually whether his local system is time- or space-rich at any given hour, so he can give simple indications to a compiler about what options should be exercised. In COBOL, for instance, all data addresses can be forced either into the operand table to minimize program storage, or in-line to speed up execution by eliminating the table indirection. Since the interpreter is already capable of decoding both forms, there is no compile-time or execution-time overhead associated with this degree of user optimization.

CONCLUSION

Defined-field design permits the definition of S-languages which are more efficient at memory utilization than contemporary machine structure. Because accessing and manipulation of arbitrarily-sized bit strings is handled automatically by B1700 hardware, various encodings may be selected solely on their inherent merit, with respect to program storage and decoding time; their suitability to the B1700 is irrelevant because the hardware is uniformly adept at manipulating all sizes of fields. One is free to choose problem representations which equalize the information content of fields in memory. Experience with compaction techniques, such as variable-length, frequency-based encodings, indicate that memory requirements can be reduced from 25 percent to 75 percent, compared to byte-oriented systems.

BIBLIOGRAPHY

1 *Burroughs B5500 information processing systems reference manual*
Burroughs Corporation Business Machines Group Sales Technical Services Systems Documentation Detroit Michigan 1964

2 P J DENNING
The working-set model for program behavior
Comm ACM 11 5 May 1968 p 323ff

3 E G COFFMAN JR T A RYAN
A study of storage partitioning using a mathematical model of locality
Comm ACM 15 3 March 1972

4 W T WILNER
Design of the Burroughs B1700
Proc FJCC72 Vol 41

5 D A HUFFMAN
A method for the construction of minimum redundancy codes
Proc IRE 40 September 1952 pp 1098-1101

6 D E KNUTH
An empirical study of FORTRAN programs
Software—Practice and Experience 1 1971 pp 105-133

7 C E SHANNON W WEAVER
The mathematical theory of communication
The University of Illinois Press Urbana Illinois 1949

Rotating storage devices as partially associative memories

by N. MINSKY

University of Minnesota
Minneapolis, Minnesota

INTRODUCTION

"Associativity" is a highly desirable property of memory devices. Unfortunately, it does not seem to fit very well into the structure of contemporary random-access memories. A realization of associativity on such memories is always involved with high density of logic, and in today's technology is bound to be very expensive. Virtually all existing implementations of associative memories are accordingly on a very small scale and are typically used for special purposes such as the support of "virtual memory" schemes. From this situation one can get the impression that large scale associative memories are impractical. Fortunately, however, it turns out that rotating memories, unlike random access memories, are very natural hosts for at least a limited degree of associative addressing.

This paper points to several latent potentialities of rotating memories, and describes a method for utilizing them for a realization of "partial associativity" (a term which is defined below).

The method described here is by no means the only possible way for realizing associativity on cylinder memories. Several related proposals were published quite recently: by Slotnick and Parker,[1,2] by Coulouris Evans and Mitchell[3] and by Gertz.[4] Other possibilities were considered by the author.[5,6] At present it is not clear to the author which of all the possible methods is preferable, and under which circumstances. This paper should be treated, therefore, as an illustration of what can be done rather than as a definite proposal.

PARTIALLY ASSOCIATIVE MEMORY—A DEFINITION

In this section we will define the term "partially associative memory" (PAM), by specifying the primitive structure of the information to be stored on it, and its operational characteristics.

(a) *The primitive information items* stored on a PAM are pairs of the form:

$$\text{item} = (n, d)$$

The components n and d will be called the *name-part* of the item and its *data-part* respectively. As we will see below, the items stored on the PAM can be addressed only through their *name-part,* which is the reason for including the word "partial" in the name given to the memory.

In the proposed realization of PAM, the name-part will be considerably smaller than the data-part. We will almost always have

$$\text{length}(n) \lesssim \text{length}(d)/10.$$

While this is quite a severe restriction it still leaves place for a wide range of applications.

(b) *The operational characteristics* of the PAM can be defined as a pair (P, I) where:

(b.1) P is a finite set of primitive predicates $(p_1, p_2, \ldots, p_k)$ defined over the name-part of the items. The following are examples of predicates which are likely to be included in P.

(1) For a given pattern of bits b, and a given mask m:

$$\text{n.AND.m} = \text{b}.$$

Here ".AND." is a masking operator and n stands again for the name-part of an item.

(2) For a given mask m, and given integers num1, num2

$$\text{num1} \leq \text{n.AND.m} \leq \text{num2}.$$

(3). A composite expression, like:

$$(\text{n.AND.m1} = \text{b}) \wedge (\text{num1} \leq \text{n.AND.m2} \leq \text{num2}).$$

(b.2) I is the set of primitive instructions that the PAM can carry out. The following instructions are considered as being essential to every PAM:

(1) Store an item (n, d).

(Note that no address is specified; the item

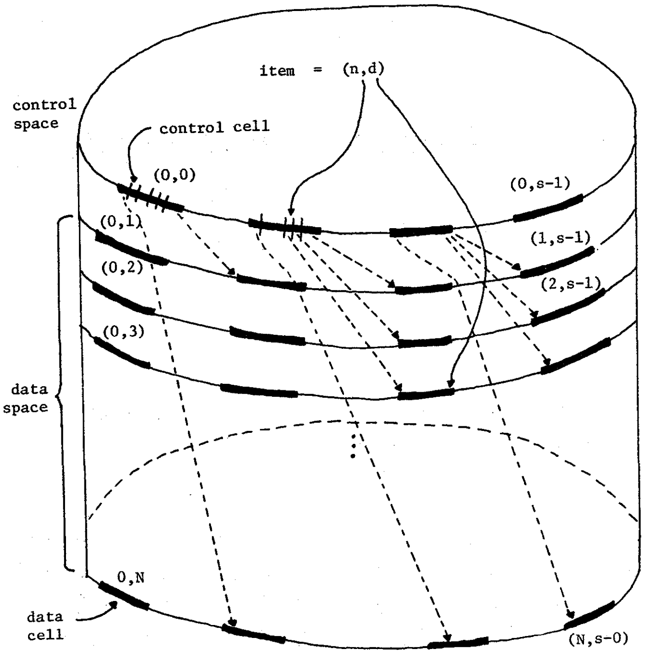


Figure 1—The partition of the cylinder storage space, and the way an item is stored on it

has to be stored in an arbitrary empty memory slot.)

(2) For a given predicate $p \in P$, retrieve an item which satisfies $p(n)$.

(3) For a given $p \in P$, retrieve the set of items satisfying $p(n)$.

(4) For a given $p \in P$ delete the set of items which satisfies $p(n)$.

A specific PAM may have more primitive instructions on top of these four, but we consider the efficient execution of the primitives defined above, as a necessary and sufficient condition for a memory to be a useful PAM.

THE SUITABILITY OF "CYLINDER MEMORIES" FOR PARTIALLY ASSOCIATIVE ADDRESSING

We will be concerned in this paper with *cylinder-memories*, either in the form of drums, or as the "cylinder part" of disks. Although the conventional structure of such memories is well-known, we will describe it here schematically, mainly in order to introduce notations which will be used in the rest of the paper, (cf. Figure 1).

The storage space of a cylinder-memory contains a set of fixed length* addressable units called pages. The pages are grouped into $N+1$ *tracks* numbered from 0 to N, and S sectors numbered from 0 to $S-1$. The pages are addressed by their track and sector indices (i, j). We will use the phrase "to access a page" for the act of reading it or writing on it.

A set of read/write heads, one head per track, is moving** together above this surface from left to right. But only one channel is serving all the heads, so that just one page can be accessed at a time.

We introduce the following notation:

T is the "revolution" time of the heads.

t_g is the amount of time required to pass over an intersector gap.

$t_s = T/S$ is the amount of time required to pass over a page plus one intersector gap.

The activity of the memory is supervised by a special processor called *controller*. In order to access a randomly given page (i, j), the controller first selects the head associated with track number i, (the head selection, being an electronic operation, is in general much shorter than t_g). Then, the controller waits until the head gets to the addressed page. During this "rotational delay" the read/write heads are passing above the cylinder while the reading mechanism is essentially looking for the addressed page, which is identified by clock pulses or by some pattern of bits.

Our proposal for a realization of associative addressing is essentially based upon an exploitation of this unavoidable delay. *Instead of spending this time in looking for a given address, we are proposing to use it for a search for a given content.*

However, with the single reading mechanism, currently available to a cylinder, it would take a long time to search its whole storage space. In principle it is possible to build a pattern matching mechanism for each head, as was proposed by Slotnick and Parker.[1,2] Such an arrangement will indeed enable full associative addressing, but it is bound to be very expensive. Having in mind standard low-cost devices we will try to achieve a less ambitious goal, namely the realization of "*partial* associative memory" as it was defined above.

Essentially, the method for achieving partial associativity is very simple: Suppose that the items to be stored on our memory satisfy the relation

$$\text{length(name-part)} \leq \text{length(data-part)}/N,$$

* The page length is not always fixed in practice but we will assume this for the sake of simplicity.

** In practice the heads are fixed in space and the cylindrical surface rotates. It is, however, easier for us to talk about the head as the moving part.

then we can group all the name parts into a single track which can be scanned with a single reading mechanism in T seconds. We will accordingly partition the storage of the cylinder as follows (see also Figure 1): The cylinder is (logically) divided into two parts, to be called *data space* and *control space.* The data space consists of N tracks (track 1 through track N), while the control space is just one track (track number 0) to be called also *control track.* The pages of the control track will be called *control pages* (CP's); a specific control page $(0, i)$ will be denoted by CP_i. We further partition the data space into equal sized cells to be called *data cells* (DC's). A data cell may be of any size depending upon the application; for the sake of simplicity, however, we assume in this paper that each CD is a page, to be also called *data page* (DP). The control space is similarly partitioned into fixed size *control cells* (CC's) so that the total number of control cells is equal to the number of data cells, which means that

$$\text{length(CC)} = \text{length(DC)}/N.$$

We now associate a unique CC with each DC as illustrated in Figure 1: The control cells in CP_i are associated with the data cells in column $i+\delta$*, while δ is a small integer which depends upon the application, as will be described later. (In Figure 1, δ is assumed to be 1). We will use the term *memory slot* for a pair (CC, DC) associated with each other.

Suppose now that the items (n, d) are stored by recording n on a CC and d on its associated DC. Given then a predicate p we can retrieve an item satisfying $p(n)$ simply by reading continuously the control track, computing p on each CC. If a CC containing an n which satisfies $p(n)$ is found, then we switch to its associated DC to read the d part of the item. The predicate p must be simple enough so that its computation can be carried out in time for the associated DC to be accessed at the same revolution; (this point will be further discussed later).

Although the above organization of information on the cylinder permits an efficient associative retrieval of an item, it still does not turn the device into a PAM, since most of the primitives of a PAM cannot be performed efficiently, as will now be demonstrated by considering two such primitives. We will see that the difficulties are due to limitations of the conventional architecture of cylinder devices, limitations which are not inherent to such memories and can be removed quite easily.

Consider first the act of *storing a new item* (n, d). This operation has three steps:

(1) An empty slot must be found. (We will assume that an empty slot is identified by a zero CC.)
(2) The control page containing the zero CC must be modified by inserting n into it.
(3) d must be written on the DC associated with the CC.

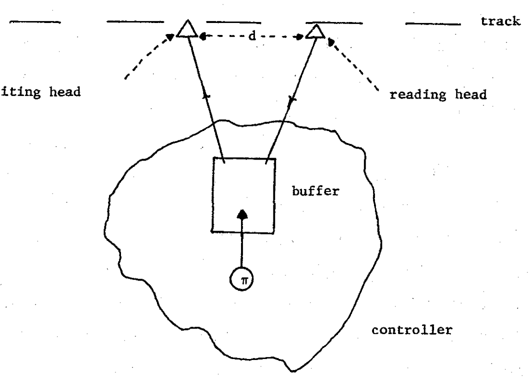


Figure 2—An illustration of the fast correction capability provided by separating between the reading and writing heads. Suppose that the heads are moving from left to right relative to the track, keeping the distance d between them constant. Correction of a page is performed as follows: The content of the page is read into the buffer by the reading head, corrected by the processor π. Then, when the writing head gets to the page, the corrected information is written on it.

When we are ready to perform step 2, after identifying an empty CC, the read/write head is not above the control page any more. We have to wait until it gets there again in order to record the corrected information. Hence, it takes more than T seconds to store an item, which does not seem acceptable.

Essentially, our problem is the inability to update a sector on conventional rotating devices in less than T seconds. This is clearly due to the fact that the same physical head is used for both reading and writing. The problem can in principle be solved by separating between the two heads, as illustrated in Figure 2.

Note that fast updating capability is very useful even for conventionally addressed memories. It is, however, crucial for the control track in the above proposed access method, since, as we will yet show, virtually all the primitive operations of a PAM require frequent modification of control cells.

There is yet another inadequacy of conventional cylinder-memories as hosts for a PAM. This is revealed by the following description of *set retrieval.*

Suppose that there are several items satisfying a predicate $p(n)$. A useful PAM must be able to retrieve

* Additions performed on the sector index are assumed to be modulo S throughout this paper.

all of them efficiently. But suppose that two of the d parts of these items happened to be written in sectors j and $j+1$. To be specific, suppose that the association between CC's and DC's is as in Figure 1. Now, the CC of the first item is sensed while the heads are above sector $j-1$; if the controller is instructed to read the associated DC, then the CC pointing to the second relevant item cannot be sensed, and we will have to wait for another revolution in order to retrieve the second item. The problem here is that we can use only one head at a time. Apparently, we need the ability to access the data space and the control space in parallel, in order to be able to retrieve sets "by content" efficiently.

In the next section we describe in detail a realization of a cylinder PAM which incorporates the above suggested modifications.

AN ORGANIZATION OF CYLINDER MEMORIES AS PAM'S

The main structural characteristics of the memory

A schematic description of the proposed memory organization is given in Figure 3. The main novel features in it are the following:

(a) We are using the partition of the storage space into *control-cells* and *data-cells,* which was introduced above. (The specific method for associating a CC to a DC is yet to be determined.)

(b) The read/write head normally associated with the *control track* is separated into two heads: the *control input head* (CIH) which reads from the control track, and the *control output head* (COH) which writes on it. (See also Figure 2.)

(c) The two heads associated with the control track have to function in parallel to each other, and to one of the data heads. There are accordingly three parallel channels connecting the controller to the device: two unidirectional channels and a bidirectional one.

(d) Apart from whatever buffer space is required for the normal data transfer, we need fairly large buffer space for manipulating the control track. The size of this *control buffer* will be determined later.

(e) To supervise the activity of the PAM we need a fairly sophisticated controller which must support five parallel activities. We will accordingly describe it as being a complex of five independent "virtual processors"* working in parallel but interacting with each other. They are listed below:

(1) The *control input channel* which reads the control track, via the CIH, into the *control buffer.*

(2) The *control output channel* which writes information from the *control buffer* into the *control track* (using the COH).

(3) The *data channel* which controls the *data heads* and transmits information from one of them (at a time) into a *data buffer* in the controller, and back.

(4) The *memory channel* which transfers information into the main (target) memory.

(5) The *monitor* which supervises the activity of the memory. (By the phrase "memory" we mean the whole complex: storage space, heads, channels, buffers, etc.)

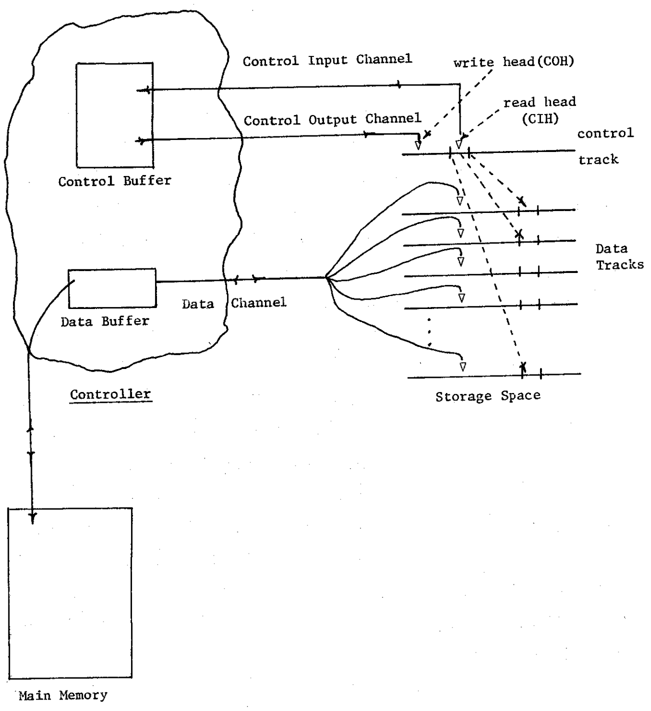


Figure 3—An illustration of the proposed memory organization (note that the cylinder storage space is illustrated in a planar form, for simplicity)

The dynamic behavior of the memory

The dynamic behavior of the memory is illustrated by the flow-chart in Figure 4. The five columns in the

* We use the term "virtual" because it should be possible to realize several of them by a single actual processor.

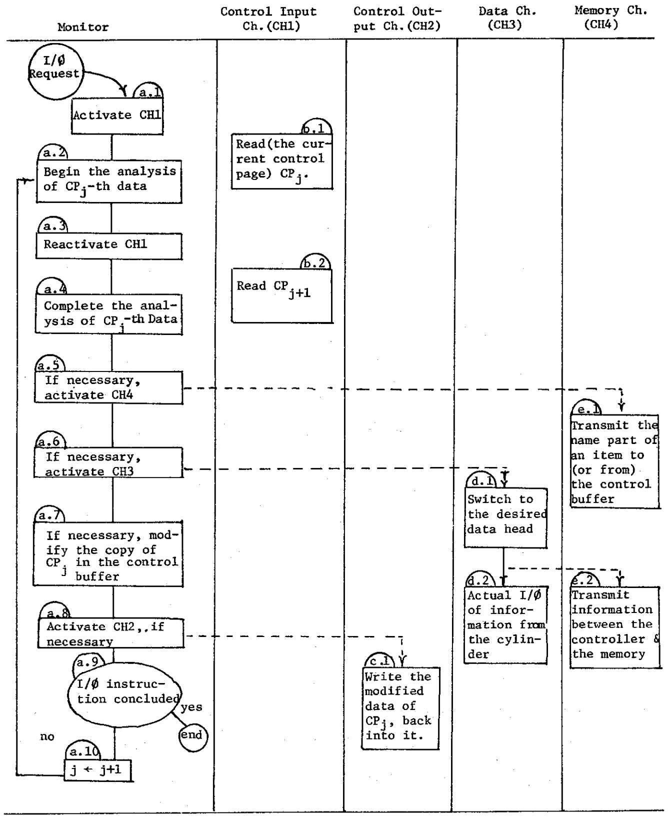

Monitor
Control Input Ch.(CH1)
Control Output Ch.(CH2)
Data Ch. (CH3)
Memory Ch. (CH4)
I/∅ Request
a.1
Activate CH1
a.2
Begin the analysis of CP$_j$-th data
a.3
Reactivate CH1
a.4
Complete the analysis of CP$_j$-th Data
a.5
If necessary, activate CH4
a.6
If necessary, activate CH3
a.7
If necessary, modify the copy of CP$_j$ in the control buffer
a.8
Activate CH2,,if necessary
a.9
I/∅ instruction concluded
yes
end
no
a.10
j ← j+1
b.1
Read(the current control page) CP$_j$.
b.2
Read CP$_{j+1}$
c.1
Write the modified data of CP$_j$, back into it.
d.1
Switch to the desired data head
d.2
Actual I/∅ of information from the cylinder
e.1
Transmit the name part of an item to (or from) the control buffer
e.2
Transmit information between the controller & the memory

Figure 4—The dynamic behavior of the controller
The five columns represent the five parallel activities of the controller. There is no significance, from the point of view of timing, to the relative location of the boxes in the various columns, see for that Figure 5

diagram represent the five processors mentioned in the previous section. The activity of the various boxes in Figure 4 depends upon the specific I/∅ instruction being served by the memory; the flow of control, however, is general. We will now describe the activity of the memory using a specific I/∅ instruction as an example. (Throughout this subsection we will ignore most of the issues involved with the need to synchronize the various parallel activities of the memory; they will be treated later in the paper.)

Suppose that the memory is instructed to store a sequence of items

$$(n_1, d_1), (n_2, d_2) \ldots (n_k, d_k).$$

The following steps must be performed for each item (n_i, d_i):

(a) An empty memory slot must be found. (We will assume that an empty slot is identified by a zero control cell.)

(b) d_i has to be written on the part of the empty slot.

(c) n_i has to be written on the control part of the slot.

The execution of this task begins by the monitor instructing the CIH to read a page from the control track (box a.1 in Figure 4). The control page so accessed is the one which happens to be nearest to the CIH at this moment; we assume it to be CP$_j$. The actual input process is represented by box b.1.

Box a.1 is executed just once per instruction; the monitor then loops between boxes a.2 and a.10, reading the control track, analyzing its information, updating it and activating the data heads and the memory channel until the I/∅ instruction is satisfied.

Box a.2 begins to analyze the data extracted from CP$_j$ (we will see later that at this moment, some, but not all of CP$_j$'s contents is transferred into the control buffer). In general, the purpose of the analysis is to verify a given predicate p on every CC of CP$_j$. p depends upon the instruction being served; in this case it is simply a search for a zero CC, which identifies an empty memory slot.

The monitor may not be able to complete the analysis of CP$_j$'s data by the time that the CIH gets close to the next sector in the track. In that case the analysis has to be interrupted in order to reactivate the CIH to continue by reading CP$_{j+1}$ (cf. boxes a.3 and b.2).

When the analysis of CP$_j$ is completed (in box a.4), the monitor is in position to decide whether an item has to be transferred into, or out of the main memory. In our case, the current item (n_i, d_i) has to be recorded on the slot identified by a zero CC, if one was found. This is done in several steps: First, n_i is written into the empty CC space of the control buffer containing the copy of CP$_j$ (boxes a.5 and c.1). This should not take more than a few microseconds as the name-part of an item is relatively small and the operation does not involve the cylinder itself. Secondly, the monitor activates the appropriate data head (boxes a.6, d.a) to access the data page associated with the empty CC found in CP$_j$; we will denote this data page by DP$_j$. The association between data cells and control cells has to guarantee that at this moment the data head will be fairly close to DP$_j$; we will return later to this point. When the data head gets to the page DP$_j$, both the memory channel and the data channel connecting the controller with the cylinder are activated (boxes d.2 and e.2).

If any additional modification to the content of CP$_j$ is required, it will be done by box a.7. In our case no such modification is necessary but it may be required in various "marking operations" as will be shown. If the buffer containing the content of CP_j was modified, either by box a.5 or by a.7, the modified buffer must be written back into CP_j modifying the page itself (boxes a.8 and c.1). Obviously, the COH must be in position to do that, which is a requirement on the physical separation between the two heads.

If the I/Ø instruction is fulfilled (box a.9), the controller idles; otherwise it goes back to box a.2 to begin analyzing CP_{j+1}'s information which by now is partly in the buffer. (Box a.10 represents the fact that now the next control page will be analyzed.)

We should keep in mind that the above description is only an example. Actually the memory must be able to carry out a variety of I/Ø instructions based upon a set of predicates. The actual activity of most boxes in Figure 4 is therefore a function of the instruction being served, and there should be a way to select the desirable algorithm each time.

One can infer from the discussion above that the controller which supervises all this activity should be fairly powerful, preferably programmable, processor. This processor must be able to perform simple computations very quickly, to support several parallel activities and to choose at run time between several alternative procedures. The price of such a minicomputer is nevertheless only a small fraction of the price of a large disk or drum. In fact there is a growing trend in the industry* to change the conventional special purpose controller to a processor which has essentially the characteristics required here.

Synchronization of the parallel activities of the PAM

In the discussion above we neglected the need to synchronize the various parallel activities of the controller with each other and with the rotation of the cylinder. Such a synchronization imposes constraints which will now be discussed.

(a) The *controller cycle* (boxes a.2 through a.10) must be executed in less than $t_s = T/S$ second^s, since it has to be applied to each of the S pages of the control track. This obviously restricts the complexity of the primitives of our PAM, for a given controller and cylinder. With the present speed of logic, however, we should have ample time at least for primitives of the type of simple pattern matching.

(b) The data pages associated with CP_j can be accessed not sooner than δ_1 seconds after the reading of CP_j is terminated, where δ_1 is defined as the time required to execute boxes a.3 through a.6 plus the "head switching" time (box d.1). This delay clearly depends upon the specific I/Ø instruction. However, for a given memory system, with its finite set of primitives we can define:

$$\bar{\delta}_1 = \max(\delta_1)$$

The delay $\bar{\delta}_1$ may be realized by an appropriate association between the control cells and data cells. There are two cases to be considered:

(1) If $\bar{\delta}_1 < t_g$ (t_g was defined as the intersector gap time), then it is enough to associate the CC's in CP_j with the data column $j+1$ (namely the pages $(i, j+1)$ for $1 \leq i \leq N$).

(2) If $\bar{\delta}_1 > t_g$, we may use several techniques: We can associate the data sector $j+2$ with CP_j or we can place the data sectors and the control pages in different angular positions. (Alternatively, the same effect may be achieved by allowing a non-zero angle between the data heads and the head reading from the control track.)

(c) The actual updating of CP_j can begin only δ_2 seconds after the reading from it was terminated. Here, δ_2 is the execution time of boxes a.3 through a.8 in Figure 4. We can again define for a specific memory:

$$\bar{\delta}_2 = \max(\delta_2).$$

The delay built into the memory must be greater than or equal to $\bar{\delta}_2$. Such a delay may be realized by an appropriate physical spacing between the read head and write head associated with the control track, so that the write head gets to every page, $\bar{\delta}_2$ seconds (or more) after the read head leaves it.

In the case of disks there is a problem involved with this delay: a fixed linear distance between the heads amounts to different time delays when the heads are accessing different cylinders. This, however, does not necessarily mean that the distance between the heads must be physically readjusted for each cylinder. It would be enough to guarantee that all the delays are greater than or equal to $\bar{\delta}_2$. The different delays can be buffered by the control buffer whose length, as we will see below, is a function of the delay.

(d) Note that during the part of the *controller-cycle* serving CP_j, we have the whole copy of CP_j residing in the control buffer while in addition the content of CP_{j+1} already flows into it. This situation exists for about $\bar{\delta}_2$ seconds, (during the execution of boxes a.4 through a.8). At the end

* Control Data Corporation, for example, now has such a controller.[7]

of this period the occupied buffer space is roughly:

$$(\bar{\delta}_2/t_s+1)\cdot(\text{page size})$$

(neglecting the intersector gap time). We do not need more buffer space than that, since after concluding the execution of box a.8, information begins to flow out of the buffer back into CP_j (box c.1) at the same rate in which it flows in, from CP_{j+1}. We thus have a rough estimate of the required buffer space. In the case of disks, if $\bar{\bar{\delta}}_2$ is the maximum of $\bar{\delta}_2$ over all the cylinders, then the required buffer space is:

$$(\bar{\bar{\delta}}_2/t_s+1)\cdot(\text{page size}).$$

An illustration of the time relationship of the various activities of the memory is provided by the time-diagram in Figure 5. It is easy to infer from this diagram, as well as from Figure 4, that the size of the delays $\bar{\delta}_1$ and $\bar{\delta}_2$ would generally be smaller, and in any case not much bigger, than t_s.

It may be instructive to present at this point an example of an actual situation. We will consider the popular CDC 841 disk file (which is virtually identical to IBM 2314). Every cylinder of this disk is constructed from 20 tracks each with 14 pages. The size of every page is 3840 bits. The revolution time T is 25ms.

In this disk we will have 19 control cells per control page; the size of each CC is about 190 bits. The *controller cycle* t_s is about 1.8 ms. which should allow a fair amount of computation to be performed on each CC. As to the size of the *control buffer*, suppose that $\bar{\delta}_2=t_s/2$ which is equivalent to saying that the distance between the read head and the write head must be at least 1.5 times the length of a page. Now, the radius of the outermost cylinder of the 841 disk file is ~6.5 inches, while the radius of the innermost cylinder is ~4.5 inches. If the two separated heads are mounted on a fixed angle fork, which has to move from cylinder to cylinder, then the biggest delay $\bar{\bar{\delta}}_2$ would be

$$\bar{\bar{\delta}}_2\approx(6.5/4.5)\cdot\bar{\delta}_2\approx0.75\cdot t_s.$$

Which means that the size of the control buffer should be:

$$(\bar{\bar{\delta}}_2/t_s+1)\cdot(\text{page size})\approx6750 \text{ bits.}$$

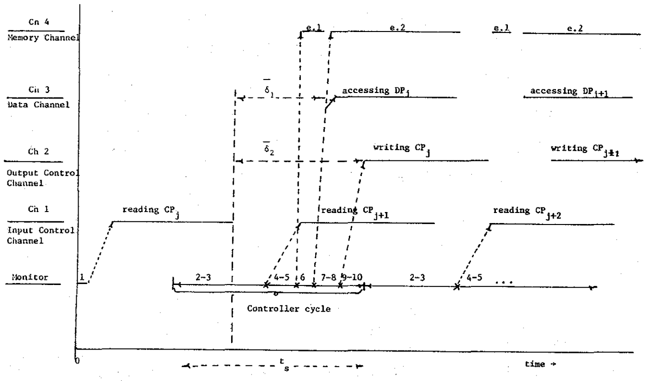


Figure 5—Illustration of the time correlation of the various parallel activities of the memory

The numbers above the "monitor line" represent the boxes of Figure 4 which are active at a specific moment. The dotted lines represent the activation of the various channels by the monitor

THE PERFORMANCE OF THE PROPOSED "PARTIALLY ASSOCIATIVE MEMORY"

We will try now to justify the name "partially associative memory" for the proposed device, by describing its performance in carrying out the operations which were defined as essential primitives of a PAM. In order to compare the performance of our PAM with that of a conventional rotating memory we will assume an *item* in the conventional case to be of page size.

Retrieval of a single item

Instead of the average $T/2+t_s$ seconds on conventional cylinder memories, it takes on the average $T/2+2\cdot t_s+\bar{\delta}_1$* seconds to retrieve an item from our PAM. Needless to say, this extra inefficiency is more than compensated for by the fact that the retrieval is "by content."

Storing a single item

This operation turns out to be almost always more efficient than its equivalent for conventional cylinder-memories. The store operation for both types of memories has two parts

(a) Getting to the point where the item has to be written;
(b) The actual writing process.

* Here, and in the rest of this section we neglect the inter-sector gap time t_g.

On conventional cylinders, the store operation takes $T/2+t_s$ seconds on the average: $T/2$ to get to the desired address and t_s for the actual writing.

To store an item on our PAM, we do not have to wait until the heads get to a prespecified sector; we can write on the first empty memory slot encountered. The time delay d involved with that, obviously depends upon the distribution of empty slots on the cylinder, and it is in general considerably smaller than $T/2$. For the case that there are e empty slots distributed uniformly, on the cylinder, the expected delay $d(e)$ was computed elsewhere.[6] Here we will bring some numerical results for illustration. If $S=10$ then we have:

e	1	5	10	15	20
expected $d(e)$ in units of t_s	4.5	1.2	0.49	0.24	0.13

These results mean that if the cylinder is not heavily loaded we have a very good chance of finding an empty slot in the first sector encountered.

On the other hand, the second part of the store operation takes longer than its conventional counterpart. Once the CIH gets to the control sector containing an empty CC, it takes $2\cdot t_s+\delta_2$ seconds to complete the store operation. This is more than the standard t_s seconds, but the much shorter rotational delay would more than compensate for it, in most cases.

Storing a set of items

This operation was already described in detail. The time required to carry it out depends again upon the distribution of empty slots on the cylinder surface. This time is calculated elsewhere[6] for the case that the empty slots are distributed uniformly.

Retrieval of unordered sets

Let $A(p)$ be the set of items defined by the simple predicate $p\in P$. Namely, $A(p)$ is the set of items stored on the PAM, which satisfy $p(n)$.

Let M be the maximal number of pages containing items from $A(p)$, which belong to a single data sector.

For the sake of retrieval of such sets (and also for other purposes) we allocate a single bit in each CC for system use; it will be called *mark-bit*. We will use the terms: to *mark* and *unmark* an item, for making its mark bit 1 and 0 respectively. In addition we will use the word *marked* as a primitive predicate which is "true" for marked items.

The following is an algorithm which retrieves the set $A(p)$ in M revolutions of the cylinder. The retrieval process has two phases:

Phase 1: During the first revolution we look for items satisfying $p(n)$. As many of them as possible will actually be retrieved and their mark bit will be set to 0 (note that not more than one item per sector can be so retrieved). All the other items satisfying $p(n)$ will be marked. Let R be the number of items so marked. R would obviously be zero if $M=1$; in this case the retrieval is accomplished in one revolution. If, however, $R\neq 0$ we continue with Phase 2.

Phase 2: Now we are looking for items which satisfy the predicate:

$$p(n)\wedge \textit{marked}$$

trying to retrieve them. (Again, we can retrieve only one per sector.) Each such item, if retrieved, is unmarked and R is reduced by one. The retrieval process is terminated when R becomes zero.

A few remarks are in order:

(a) The marking of items does not take any extra time; this is due to the efficient correction capability of the control space provided by the head separation.

(b) The algorithm can be carried out only if the memory can handle predicates of the form:

$$p(n)\wedge \textit{marked}.$$

Note, however, that if we can guarantee that items which do not belong to $A(p)$ are unmarked, then in phase two we can simply look for marked items.

(c) The efficiency of set retrieval obviously depends upon the distribution of the set on the cylinder surface. This means that although we do not have to know the address of an item stored on a PAM in order to retrieve it, we should keep some control on the distribution of sets of items. The efficiency of this operation for randomly distributed sets is calculated elsewhere.[6]

Deletion

To delete an item, it is enough to erase its control cell. Given a simple predicate p we can therefore delete the set $A(p)$ in $T+t_s$ second[s] regardless of its distribution on the cylinder. This efficiency is again due to the head separation.

Other associative I/Ø operations, such as the retrieval of sets defined by composite predicates are discussed elsewhere.[5,6]

Some odd applications of the proposed memory

Although a comprehensive study of the applications of the proposed memory is not within the scope of this paper, we will mention here two of its less obvious usages. The usages to be discussed here depend more on the effectively *instantaneous modification capability* created by the head separation, than on the associative addressing.

(a) *Synchronization of parallel processes* which access the same information, is a common problem in data-base management. In particular, if two processes $P1$ and $P2$ are capable of modifying the same item i, they must be prevented from doing that at the same time. Otherwise, the following undesirable sequence of events may occur:

$P1$ reads item i.
$P2$ reads item i.
$P1$ replaces i with modified information.
$P2$ replaces i with modified information.

In the context of the proposed PAM, such processes can be synchronized by means of "semaphores" (cf. Dijkstra[8]). Every bit in the control cell can serve as a semaphore since it can be set and reset instantaneously.

(b) The second useful application of the "instantaneous modification capability" is that it provides us with a very simple and cheap way for *maintaining various "usage statistics."* Suppose that we are interested in the number of retrievals of each item in the memory. Suppose in addition that we can afford to allocate a suitable count field in each CC. All we have to do then is to increment by one the count field of every item when retrieved. This capability may be quite useful for data-base administration.

CONCLUSION

As pointed out in the introduction, the detailed description of an organization of rotating associative memory was not intended to imply that it is the only, or even the best way for realizing associativity on rotating devices. The basic ideas presented in this paper can be put together in several different ways as it is shown elsewhere.[5,6] One of the possible variations is worth mentioning here: The control information can be stored on a relatively slow random-access memory, external to the rotating device. *Such an organization does not require any change in the device itself.*

Space limitations do not allow us to include any discussion of the applications of the proposed memory. Such a discussion is particularly necessary because of two reasons:

(a) Because of the current unavailability of large scale associative memories, there is almost no published study of their use. (There are, however, several papers which discuss the use of *simulated* associative memories in data-base environment.[9])

(b) The proposed memory is quite restricted; it is only "partially associative" and it is not really large, from the point of view of "data base" type applications. Therefore, the ways for utilizing such memories are not obvious.

Elsewhere[6] we consider some applications of the proposed PAM, but a comprehensive study of the subject is yet to be done.

ACKNOWLEDGMENT

It is my pleasure to thank Dr. Peter Patton and Dr. Larry Kinney for reading drafts of this paper and for their useful comments and suggestions.

REFERENCES

1 D L SLOTNICK
Logic per track devices
Advances in Computers Academic Press 1970

2 J L PARKER
A logic per track retrieval system
IFIP Congress 1971

3 G F COULOURIS J M EVANS R W MITCHELL
Towards content-addressing in data bases
The Computer Journal Vol 15 No 2 1972

4 J L GERTZ
Storage reallocation in hierarchical memories
Third Symposium on Operating Systems Principles October 1971

5 N MINSKY
Rotating storage devices as "partially associative memories"
University of Minnesota Computer Sciences Department Technical Report 72-4 April 1972

6 N MINSKY
On associative addressing in cylinder memories
To be published

7 *A nonformal description of CDC/844 disk system*

8 E W DIJKSTRA
Cooperating sequential processes
Programming Languages F Genuys (ed) Academic Press 1968

9 J A FELDMAN
An algol-based associative language
Communications of the ACM August 1969

The page fault frequency replacement algorithm*

by WESLEY W. CHU and HOLGER OPDERBECK

University of California
Los Angeles, California

INTRODUCTION

Dynamic memory management is an important advance in memory allocation especially in virtual memory and multiprogramming systems. In this paper we consider the case of paged memory systems: that is, the physical and logical address space of these systems is partitioned into equal size blocks of contiguous addresses. The paged memory system has been used by many computer systems. However, the basic memory management problem of deciding which pages should be kept in the main memory to allow efficient operation without wasting space is still not sufficiently understood and has been of considerable interest. Obviously, pages should only be removed from the main memory if there is a very low probability that they will be used in the near future. The difficulty lies in trying to determine which pages to remove, without incurring difficult implementation problems at the same time.

Many replacement algorithms have been proposed and studied in the past, such as Random, First-in First out, and Stack Replacement Algorithms[1] [for example, Least Recently Used (LRU)]. These replacement algorithms are usually operated with a fixed size memory allocation. For such a fixed size memory replacement algorithm we need to have prior knowledge about program behavior. For example, in the LRU case, we need to have an estimate for the number of page frames which have to be allocated for each individual program. Further, program behavior is usually data dependent and changes during execution. An efficient replacement algorithm should therefore automatically adapt to the dynamically changing memory requirements. A recent study by Coffman and Ryan[2] shows that such dynamic storage partitioning provides substantial increases in storage utilization over fixed partitioning. The working set model of program behavior[3] takes into account the varying memory requirements during execution. With respect to this model, we call the replacement algorithm, which keeps exactly those pages in main memory that have been accessed during the last τ references, the Working Set Replacement Algorithm. The performance of the Working Set Algorithm still depends on the choice of the working set parameter τ and program characteristics (e.g., locality).[4] Further, the Working Set Algorithm appears expensive to implement. Therefore we were motivated to develop an adaptive replacement algorithm which is largely independent of program behavior and input data and is simple to implement. We shall use the page fault frequency (the frequency of those instances at which an executing program requires a page that is not in main memory) as an adaptive parameter to control the decision process of the replacement algorithm. Since the page fault frequency reflects the actual memory requirements of a program at execution time, the Page Fault Frequency (PFF) Algorithm can be applied to arbitrary programs without prior knowledge about program behavior.

The performance of replacement algorithms is usually compared in terms of efficiency and space-time product. Because of the complex nature of program behavior, we used simulation techniques to measure the efficiency and the space-time product for various programs. From these simulations we were able to compare the performance of the LRU and Working Set Replacement Algorithms. Next we describe the PFF Algorithm and compare its performance with the LRU and Working Set Replacement Algorithms. Finally, we discuss the advantages of this new replacement algorithm when employed in a multiprogramming environment and the implementation of the PFF Replacement Algorithm.

PERFORMANCE OF LRU AND WORKING SET REPLACEMENT ALGORITHMS

Because of the complex nature of program behavior, analytical estimation of such parameters as page fault

* This research was supported by the U.S. Office of Naval Research, Mathematical and Information Sciences Division, Contract Number N00014-69-A-0200-4027, NR 048-129.

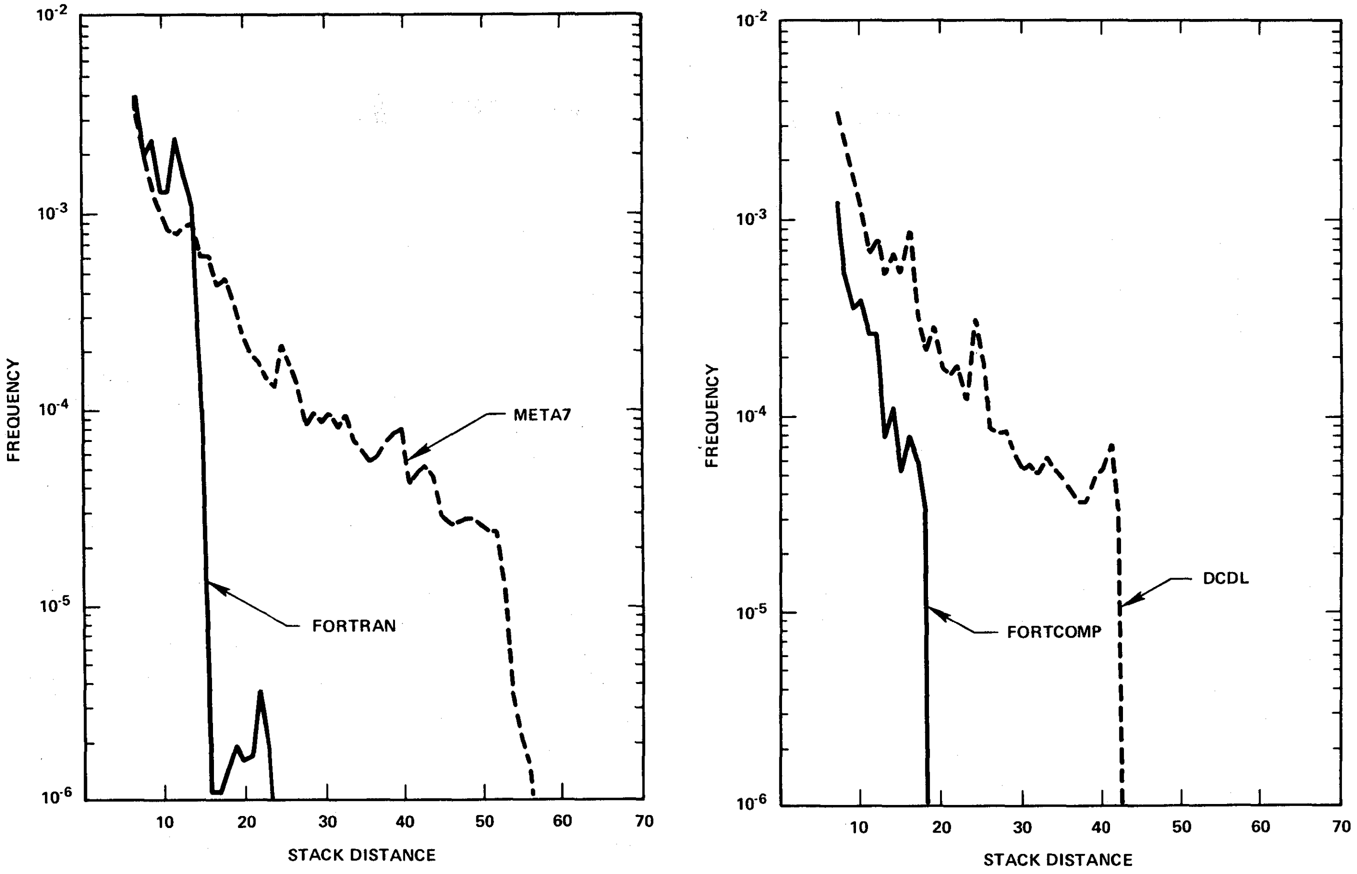


Figure 1—Stack distance frequency for the four measured programs
a) FORTRAN and META7

b) FORTCOMP and DCDL

frequency and the average inter-page-fault-time (average process running time between page faults) becomes very difficult. Yet this information is important in the planning of an efficient replacement algorithm that optimizes system performance. Therefore we employ measurement techniques for such estimations. This technique has been used previously to measure dynamic program behavior[5] and also to measure the performance of the Belady Optimal Replacement Algorithm,[6] the LRU Replacement Algorithm,[7,8] and the Working Set Replacement Algorithm.[4]

For this purpose an interpreter for the UCLA SIGMA-7 time-sharing system has been developed. This interpreter is capable of executing SIGMA-7 object programs by handling the latter as data and reproducing a program's reference string. This sequence, in turn, can then be used as input to programs which simulate various types of replacement algorithms. For convenience in presentation, we let the time required for a thousand page references correspond to one millisecond (msec).

Four different programs of various characteristics were interpretively executed. A FORTRAN Program (FORTRAN) and a FORTRAN compiler (FORTCOMP) were chosen as representatives for programs with small localities. A META7 compiler and a DCDL compiler represent programs with large localities. META7 translates programs written in META7 to the

TABLE I—Characteristics of Measured Programs

	SIZE		NUMBER OF PAGE REFERENCES
	STATIC s_o	DYNAMIC r_o	
FORTRAN	24	38	4,870,000
FORTCOMP	24	39	3,810,000
DCDL	44	71	3,010,000
META7	84	165	2,590,000

assembly language of the SIGMA-7. The DCDL (Digital Control and Design Language) is written in META7. It translates specifications of digital hardware and microprogram control sequences into machine code. To illustrate the behavior of these programs, Figures 1a and 1b display the stack distance frequencies as defined in Reference 1. The frequent occurrence of large stack distances (20 and more) for META7 and DCDL indicates that the localities for these programs are larger than the localities of FORTRAN and FORTCOMP.

Table I shows some characteristic properties of these programs. The column 'size' is divided into two parts. 'Static' refers to the number of pages, s_0, necessary to store the program as an executable file on a disk where one page consists of 512 32-bit words. 'Dynamic' indicates the number of different pages, r_0, actually referenced while processing the given input data. There are two reasons why r_0 is not equal to s_0: first, not all the pages which make up the program may be referenced while processing a particular set of input data; second, a number of data pages is created and accessed during execution to provide for working storage space, buffer areas, etc.

The number r_0 is of special interest because it is equal to the minimal number of page faults which will be incurred by every replacement algorithm based on demand paging. Actually, r_0 page faults will occur even if not a single page is replaced. In this case, all page faults are caused by the very first reference to a page.

For a given page reference string ω and a given replacement algorithm with its parameter, the page fault frequency $f(\omega)$ is defined as the ratio of the number of page faults during processing ω to the total number of references in ω.

$$f(\omega)=\frac{r}{t}$$

where

r = total number of page faults

t = total number of page references

For a finite t, since $r \geq r_0 > 0$, $f(\omega)$ is always greater than 0. The average inter-page-fault-time is t/r page references.

If no pages are replaced, the smallest page fault frequency is $f_0(\omega)=r_0/t$ which depends on r_0 and t. In general, $f_0(\omega)$ is different for different programs and could be different even for the same program when processed with different sets of input data. For this reason, it is awkward to use $f(\omega)$ as a measure to compare the performance of a replacement algorithm when applied to different reference strings. We therefore define a normalized page fault frequency, $f_n(\omega)$, as

$$f_n(\omega)=\frac{r-r_0}{t}.$$

The normalized page fault frequency considers only those page faults which are caused by references to pages which have been accessed before but which were replaced later. Clearly, if no pages are replaced, $f_n(\omega)$ is 0.

The efficiency E for a program execution is defined as the ratio of total virtual processing time (processing time without page-wait times) to the total real processing time (total virtual processing time plus total page-wait times); that is,

$$E=\frac{\text{total virtual processing time}}{\text{total real processing time}}$$

$$=\frac{1}{1+f(\omega)\cdot R}, \tag{1}$$

where

$R=T_s/T_m$

T_m = access time of main memory

T_s = access time of secondary memory

R is called the speed ratio of a particular combination of secondary and main memory. We assume T_m to be the time of one page reference (10^{-3} msec). The maximum efficiency which can be achieved if no pages are replaced is

$$E_0=\left(1+\frac{r_0}{t}\cdot R\right)^{-1}$$

E_0 again depends on r_0 and t and is therefore in general different for each reference string. For this reason, we define the normalized efficiency E_n which corresponds to the normalized page fault frequency $f_n(\omega)$, as

$$E_n=[1+f_n(\omega)\cdot R]^{-1}$$

and use this as a measure to compare the performance of various replacement algorithms. Note that E_n always reaches its maximum of 100 percent if no page is replaced that will be referenced again, and that E_n is independent of r_0.

Figures 2 and 3 display the normalized efficiency and the average inter-page-fault-time as a function of memory space allocation for the LRU Algorithm with $R=10{,}000$.* For a given memory space we notice that

* These curves can be derived directly from the stack distance frequencies in Figure 1 (see Reference 1).

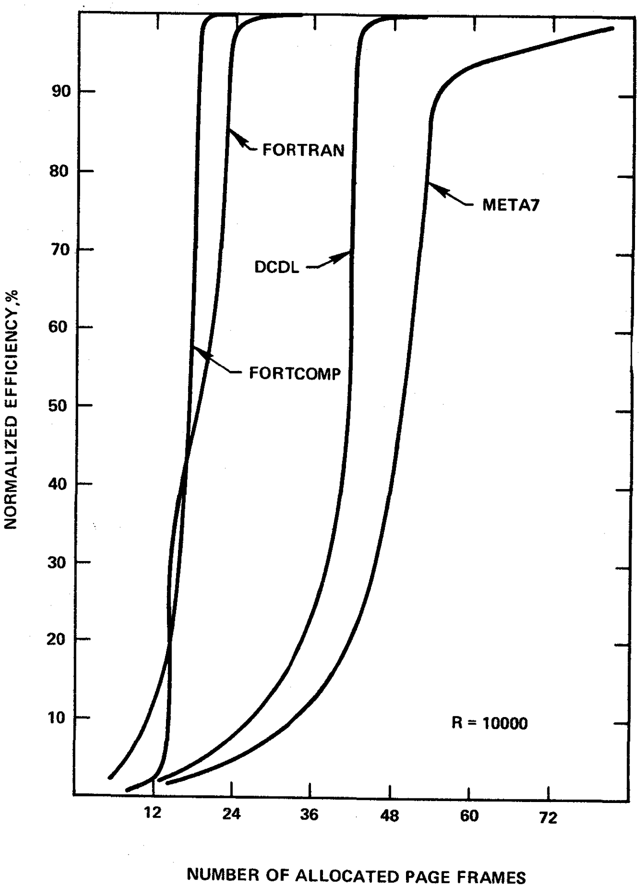


Figure 2—Normalized efficiency of the LRU algorithm

different programs have different normalized efficiencies and different average inter-page-fault-times. Further, programs with small localities tend to yield better performance than programs with large localities. The average inter-page-fault-time increases as the assigned memory space increases and reaches its maximum, t/r_0, as the memory space reaches a certain size. At this memory size the normalized efficiency reaches 100 percent. Further increase in memory space does not increase the average inter-page-fault-time and efficiency. The fact that all four curves in Figures 2 and 3 have their steepest slope occurring at different memory sizes reflects the different memory needs for each program. Thus, for a given process that uses the LRU replacement algorithm, one of the most difficult tasks is to determine the size of the memory which is to be allocated for each process. Assigning too large a number of page frames for a process results in inefficient utilization of memory space, while assigning too small a number of page frames yields too many page faults, resulting in inefficient operation. A procedure which gives the same amount of main memory to every process will almost surely result in either inefficiencies or waste of storage. In addition, the estimate of the memory needs of a process should be fairly accurate because only a few pages less than actually necessary means, in many cases, a large decrease in the average inter-page-fault-time. The determination of the amount of required memory is further complicated by the fact that it is usually data dependent and may vary during execution.

In contrast with the LRU Algorithm, the Working Set Replacement Algorithm requires a variable sized storage space for each process. This variable storage space provides the capability to adapt to dynamic changes in program behavior. The working set $W(t, \tau)$ at a given time t is the set of distinct pages referenced in the process (or virtual) time interval $(t-\tau+1, t)$, that is, the set of pages accessed during the last τ references where τ is called the working set parameter. The working set size $w(t, \tau)$ is the number of pages in $W(t, \tau)$. The basic replacement policy is to keep in the main memory those pages which have been referenced

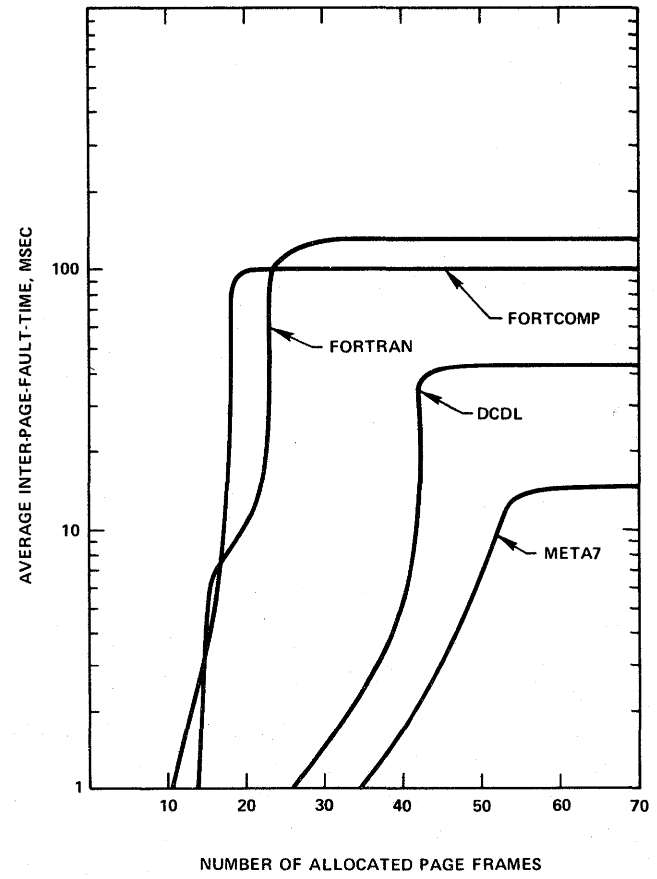


Figure 3—Average inter-page-fault-time of the LRU algorithm

during the last τ msec. τ is an important parameter which affects the performance of the Working Set Algorithm.

In general, the Working Set Algorithm can be considered as an LRU Algorithm with variable size memory allocation. There is, however, a crucial difference. Using an LRU Algorithm, pages are always replaced when a page fault occurs. This does not apply to the Working Set Algorithm. Here, page frames are freed whenever they have not been accessed during the last τ msec. A strict implementation would require the setting of a flag at this point; that is, an indication that the corresponding page frame can be used for a different page of any process. Another problem is to detect the exact time when a page has not been referenced during the last τ msec. Hence, it appears to be rather expensive to implement the Working Set Algorithm.

In the simulation of the Working Set Algorithm we assume that exactly the working set is kept in main memory. This means that a page fault is recorded whenever a page is accessed which is not included in the current working set. However, in systems using the working set strategy it may happen that a page leaves the working set and returns later without leaving main memory in the meantime. Thus, the actual page fault frequency during execution can be slightly lower than the page fault frequency we observed during simulation. This, of course, is true for every replacement algorithm which frees page frames without allocating them immediately to another page.

Figures 4 and 5 display the normalized efficiency and the average inter-page-fault-time as a function of τ for the four programs when the Working Set Replacement Algorithm is employed.* Both the average inter-

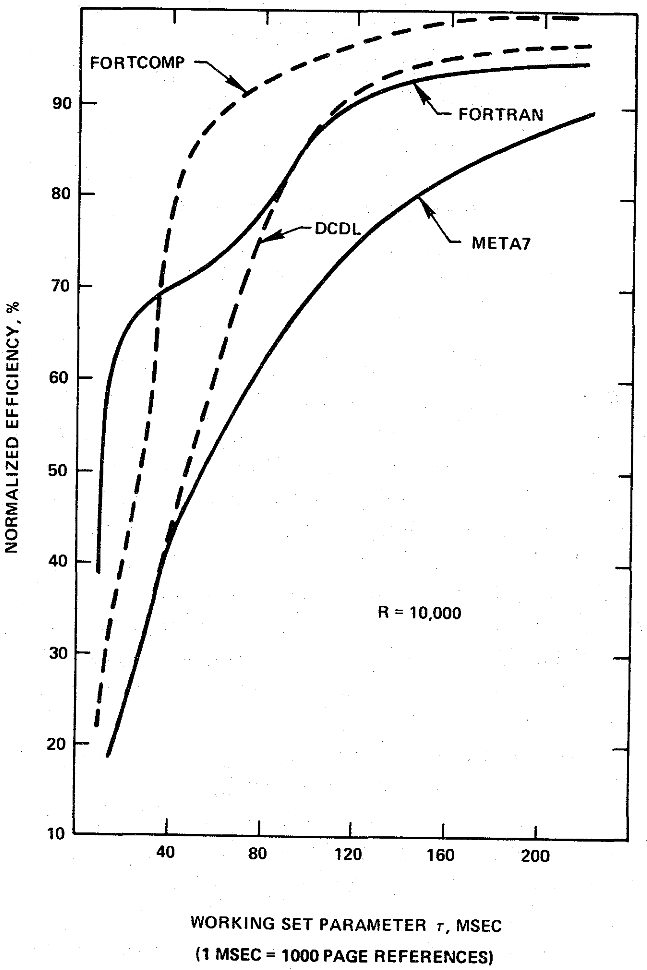


Figure 4—Normalized efficiency of the working set algorithm

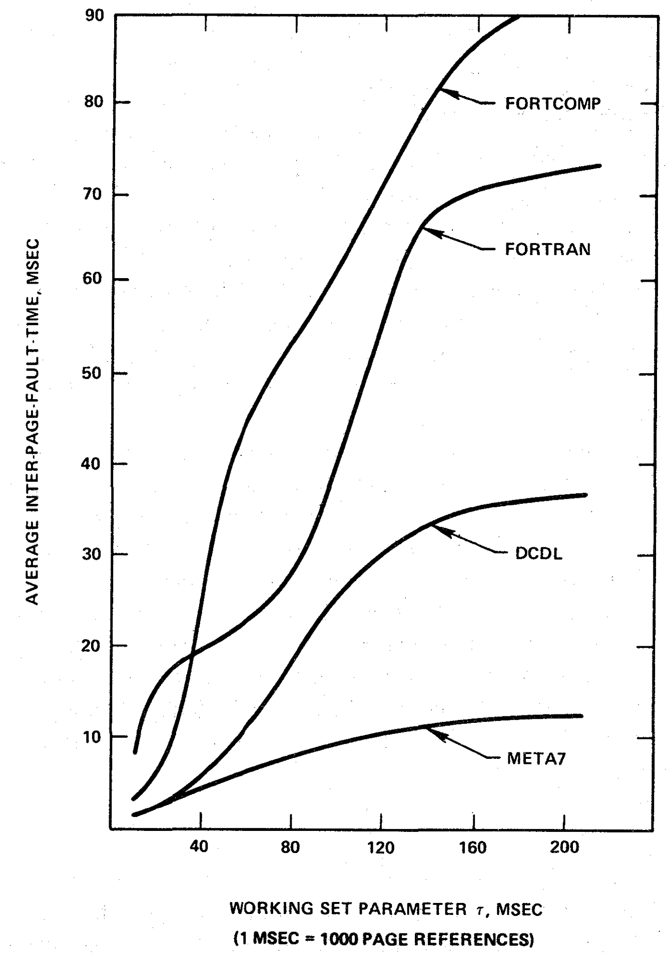


Figure 5—Average inter-page-fault-time of the working set algorithm

* For more measurement data on the Working Set Replacement Algorithm and their applications, interested readers should refer to Reference 4.

page-fault-time and the normalized efficiency increases rapidly as τ increases for the range of τ less than 80 msec. This suggests that in implementing a Working Set Replacement Algorithm for these four measured programs, we should choose a parameter τ of at least 80 msec. For $\tau = 200$ msec, the normalized efficiency for all four programs is greater than 85 percent. We also note that different programs achieve different efficiencies. For a given τ, the programs with small localities usually give better performance than the programs with large localities. This occurs because τ may not be large enough to keep the favored pages in the main memory to assure a high efficiency and high average inter-page-fault-time.

As defined in (1), program efficiency is the ratio of total virtual processing time to the total real processing time and provides us with information on how fast the process will run. From an economic point of view, another parameter of interest is cost. Therefore, we shall also include cost of storage as a measure to compare various replacement algorithms. One of the criteria is the space-time product which can be considered as being proportional to the cost of storage. Belady and Kuehner[9] define the space-time product C during the real time interval $(0, t)$ as

$$C = \int_0^t S(z)\,dz \qquad (2)$$

where $S(z)$ is the amount of storage occupied by the process at time z. The real time occupancy of information in main memory can be much longer than the actual processing time. This occurs because of multiple processes running in parallel and because of page-wait times. Since our study is concerned with the application of replacement algorithms to individual programs, only page-wait times have been considered.

If we consider the execution of a program as a discrete process, the integral in (2) can be replaced by a sum which consists of two parts. The first part is the space-time product, due to the actual processing, while the second part is due to the total page-wait time. Thus, the space-time product C can be re-written as

$$C = \sum_{i=1}^{t} S_i T_m + \sum_{i=1}^{r} S_{t_i+1} \cdot R \cdot T_m \qquad (3)$$

where t is the total number of references; r is the total number of page faults; S_i is the number of allocated page frames prior to the ith reference (i is called the number of the reference); t_i is the number of the reference which causes the ith page fault (since we do not preload any pages, $t_1 = 1$); and S_{t_i+1} is therefore the number of page frames which are allocated during the ith page-wait time.

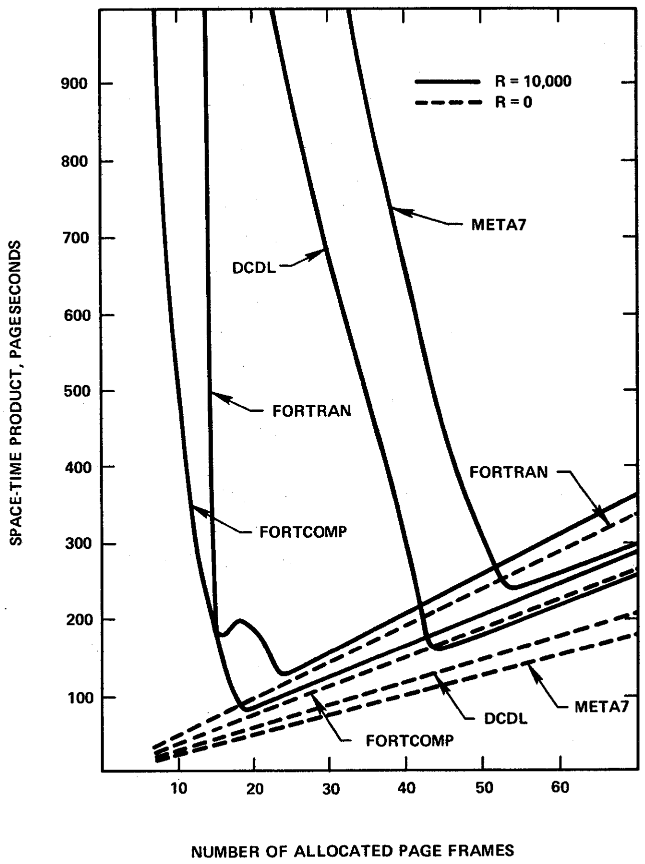


Figure 6—Space-time product of the LRU algorithm

Since $\sum_{i=1}^{r} S_{t_i+1} \cdot T_m$ and $\sum_{i=1}^{t} S_i T_m$ are independent of R, C is a linear function of R. If we know C for $R = 0$ and C for another $R (0 < R < \infty)$, then we can compute $\sum_{i=1}^{r} S_{t_i+1} \cdot T_m$ from (3), and thus C for any R for $0 < R < \infty$. For example, the space-time products of $R = 0$ (the dashed lines) and $R = 10{,}000$ for META7 with the LRU Algorithm are shown in Figure 6. The space-time product for $R = 5{,}000$ can be obtained easily from a linear interpretation of the curve $R = 0$ and $R = 10{,}000$. Hence the space-time product presented in this paper can also be extrapolated to other values of $R (0 < R < \infty)$.

Figure 6 displays the space-time product as a function of the allocated memory space for the LRU Replacement Algorithm with $R = 10{,}000$. Figure 7 displays the space-time product as a function of τ for the Working Set Replacement Algorithm with $R = 10{,}000$.

Comparing Figure 6 with Figure 2, we observe two interesting properties. First, the minimum space-time product is below the number of page frames necessary to achieve maximal efficiency. This means that minimal space-time product and maximal efficiency can be

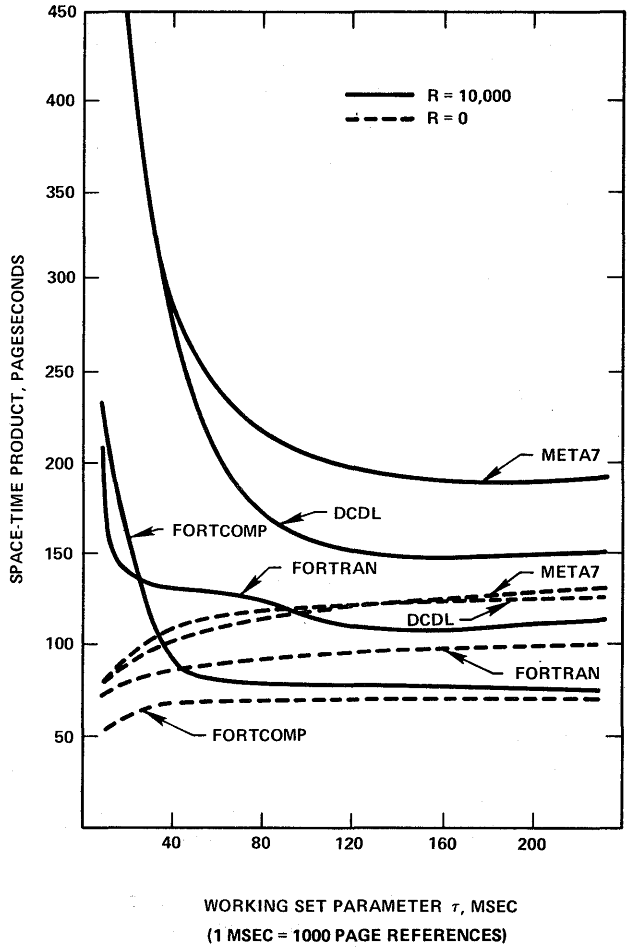


Figure 7—Space-time product of the working set algorithm

contradictory. This is because, when the number of allocated page frames reaches the point that yields the minimal space-time product, an additional page frame may decrease the total number of page faults (that is, increase the efficiency) and thereby possibly decrease the space-time product due to page-wait times [second term in (3)]. However, this decrease is not off-set by the increase in the space-time product due to actual processing [first term in (3)].

Second, in terms of the space-time product it is much more disastrous not to allocate enough pages to a process than to allocate too many. For instance, in the META7 case, if 70 pages of memory space are allocated instead of the optimal number of 55 pages, the space-time product increases from 247 to 306 pageseconds. But if only 40 pages are assigned to the same process, the space-time product increases to 700 pageseconds. Comparing Figure 7 with Figure 4, we observe similar characteristics except that the Working Set Replacement Algorithm has a much better and less sensitive (with respect to τ) space-time product than does LRU.

From the above study we know that the major disadvantage of the LRU Replacement Algorithm is that it is not at all clear how many pages have to be allocated for different programs in order to assure efficient running without wasting space. In addition, this number is usually data dependent and may vary during execution. The Working Set Replacement Algorithm constitutes a possible solution to this problem. In this case, the amount of main memory which has to be allocated is automatically determined by the number of pages referenced during the last τ msec.

Since all of the measurements were done with the SIGMA-7 time-sharing system, the results presented here are somehow characterized by this very system. However, the conclusions are likely to be applicable to other paging systems. The reason for this is that program behavior is mainly determined by programming characteristics as for example, looping, modularity of code, and data layout. These characteristics are relatively independent of, for instance, word length or instruction set, and are common to all major computer systems. This assumption is supported by the fact that the results of the LRU Algorithm simulations for the SIGMA-7 are very similar to those for other computer systems (see Joseph's results for the ATLAS-computer,[8] and Coffman and Varian's results for the IBM 360/50[7]).

From the above measurement results, we know that a replacement algorithm that is to achieve a small space-time product, has to adapt itself to the memory requirements at execution time. Therefore, the number of page frames assigned to a process must not be a constant, but must vary according to program demand during execution. Lastly, the implementation of the replacement algorithm should be simple and of low cost. With these as our objectives, in the next section we introduce a new replacement algorithm that has these properties.

THE PAGE FAULT FREQUENCY REPLACEMENT ALGORITHM

An "ideal" replacement algorithm should be independent of prior knowledge about program behavior; instead, all of the information needed to assure efficient memory allocation should be gathered during program execution. Conceptually, the Working Set Algorithm accomplishes this by keeping a table with an entry for each page which indicates when the corresponding page has been referenced last. This information enables the memory scheduler to decide when a page frame can be freed.

The Page Fault Frequency Algorithm uses the measured page fault frequency as the basic parameter for the memory allocation decision process. We assume that a high page fault frequency indicates that a process is running inefficiently because it is short of page frames. A low page fault frequency, on the other hand, indicates that a further increase in the number of allocated page frames will not considerably improve the efficiency and, in fact, might result in waste of memory space. Therefore, to improve system performance (e.g., space-time product) one or more page frames could be freed.

The basic policy of the PFF Algorithm is: whenever the page fault frequency rises above a given critical page fault frequency level P, all referenced pages which were not in the main memory, therefore causing page faults, are brought into the main memory without replacing any pages. This results in an increase in the number of allocated page frames which usually reduces the page fault frequency. On the other hand, once the page fault frequency falls below P, page frames may be freed. The same operation will be repeated whenever the page fault frequency rises above P again. We shall designate P, measured in number of page faults per msec (1 msec = 1,000 page references) as the PFF-parameter. P may be expressed as $P=1/T$, where T is the critical inter-page-fault-time. We shall use the inversion of the inter-page-fault-time as a running estimate of the page fault frequency. That is, at the time of a page fault, if the inter-page-fault-time is less than or equal to T, then we increase the number of page frames by one and no pages in the main memory are replaced. For example, $P=1/50$ means that if one or more page faults (excluding the current one) occurred during the last 50 msec, the memory scheduler will add one page frame to the allocated memory at the time of a page fault. As an initial condition it is assumed that a page fault occurred at time 0. This allows a process to start collecting its pages at the beginning of processing. Note that this "increase decision" is based only on the number of page faults which occurred during the last T msec (virtual time) and is not based on the total number of page faults which occurred since the processing began. Thus the PFF Replacement Algorithm provides very fast response to a sudden increase in memory requirements.

Let us now describe the "decrease decision" rules of the PFF Algorithm: (1) page frames in the main memory are only freed at the time of a page fault, (2) only those page frames are freed which have not been referenced since the last page fault occurred, and (3) page frames are freed only if the page fault frequency lies below the critical level P. More precisely, let t_k denote the time at which the kth page fault occurred. The page fault frequency lies below the critical level P at time t_{k+1} if and only if $t_k<t_{k+1}-T$, where $T=1/P$. In this case, all page frames which have not been accessed in the time interval (t_k, t_{k+1}) are freed. If $t_k \geq t_{k+1}-T$, then no page frame is freed since the page fault frequency lies above P. We note that T is somewhat similar to the working set parameter τ. However there is an important difference. While τ indicates when a page should be freed, T represents only a lower limit. Furthermore, in contrast to the Working Set Algorithm,* page frames in the main memory are only freed at the time of a page fault. Therefore, the time at which a page frame is freed depends not only on T but also on the page fault frequency. This policy provides the PFF Replacement Algorithm with an extra adaptive capability which makes its performance less dependent on program characteristics than is the case with the Working Set Algorithm. The PFF Replacement Algorithm may therefore be considered as a Working Set Algorithm with variable τ, where the value of τ is determined by

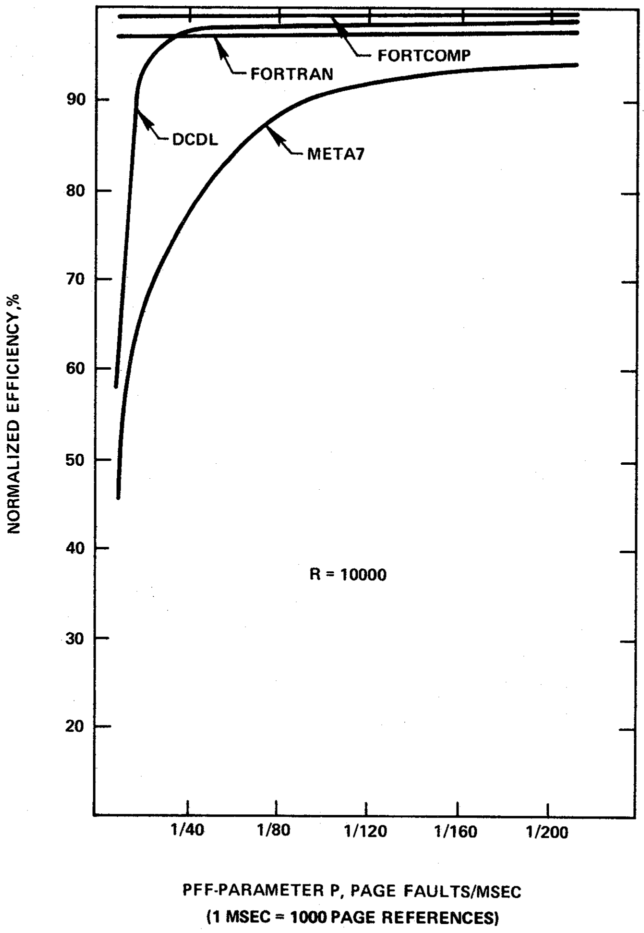


Figure 8—Normalized efficiency of the PFF algorithm

* This is, of course, only true for the strict implementation of the Working Set Algorithm which has been simulated.

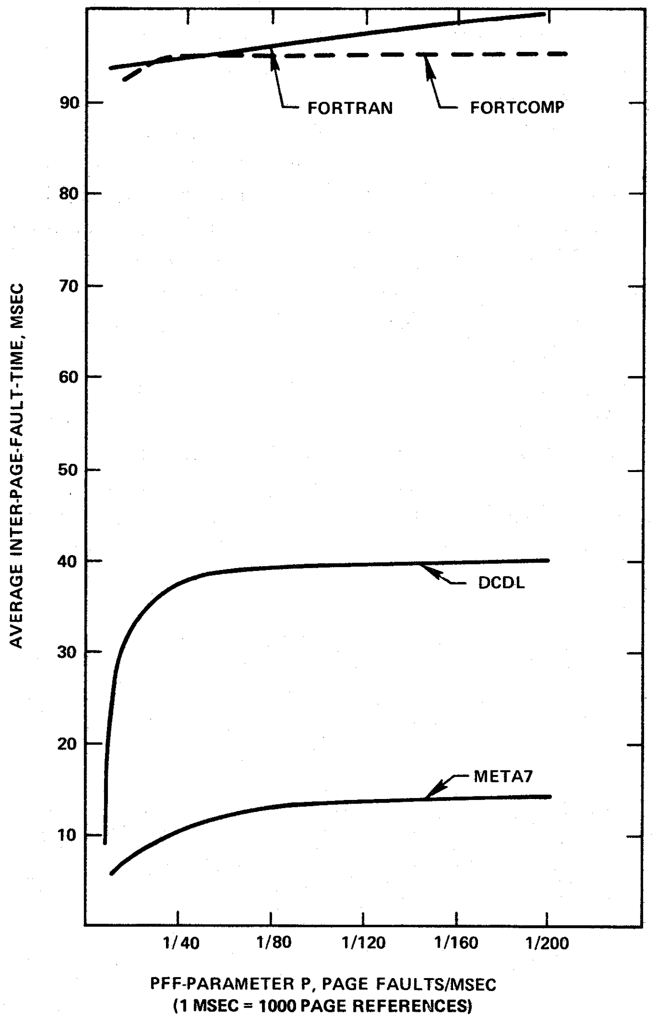


Figure 9—Average inter-page-fault-time of the PFF algorithm

the page fault frequency and the lower limit of τ is $T=1/P$. The PFF Algorithm may also be viewed as a LRU Replacement Algorithm with variable size memory allocation where the size is determined by T and the inter-page-fault-times.

Figures 8 and 9 show the normalized efficiency and the average inter-page-fault-time for the PFF Replacement Algorithm for which P ranges from 1/10 to 1/200 page faults/msec. Figure 8 reveals two interesting properties of the PFF Algorithm: (1) For $P<1/100$, the normalized efficiency is larger than 90 percent for the four measured programs. This implies that high efficiency can be achieved by the PFF Replacement Algorithm by using the same page fault frequency parameter for all four programs regardless of their size and characteristics. (2) for FORTCOMP and FORTRAN the normalized efficiency is virtually independent of P. The same is true for DCDL if P is less than 1/40 and for META7 if P is less than 1/120 page faults/msec.

Figure 10 displays the space-time product for the PFF Replacement Algorithm. Again we can observe that the performance of the PFF Algorithm is almost independent of the choice of P for $P<1/50$. For the four measured programs, the space-time products are almost constant over a wide range of values of P. The performance of the PFF Replacement Algorithm is therefore relatively insensitive to P. This is a very appealing feature of the PFF Algorithm since it alleviates the task of selecting a parameter P for implementation.

In Figure 11, the number of allocated page frames is displayed as a function of virtual processing time. As can be seen, the memory requirements for the four programs are quite different, and the number of allocated pages varies during execution, particularly for META7 and FORTRAN. This clearly demonstrates the adaptive capability of the PFF Algorithm. The area

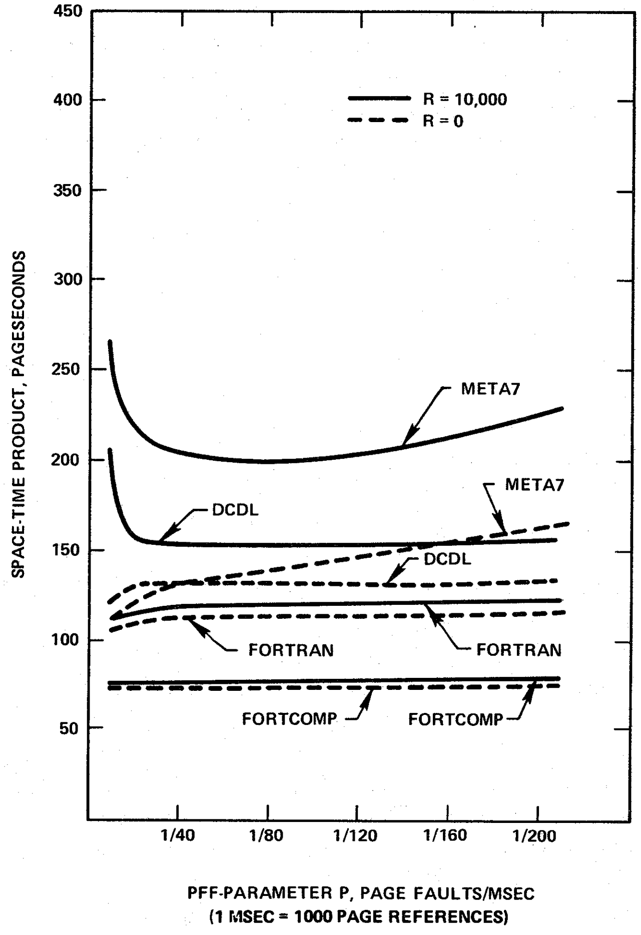


Figure 10—Space-time product of the PFF algorithm

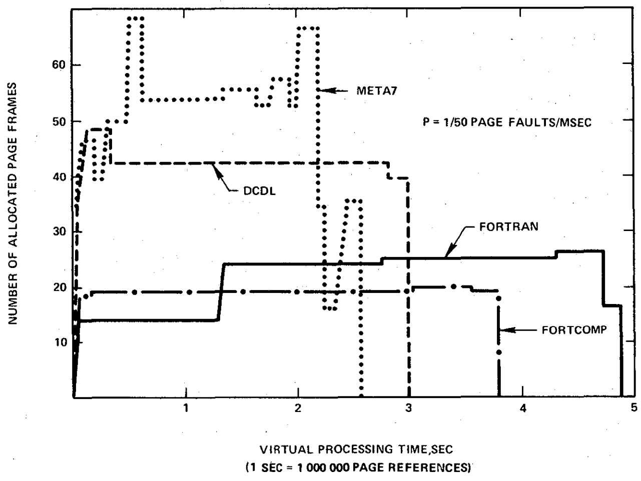


Figure 11—Dynamic changes in memory allocation of the PFF algorithm

below the four curves corresponds to the space-time product due to actual processing time. For simplicity in representation, only major changes in the number of allocated page frames are indicated in Figure 11. Nevertheless, the figure shows clearly that the majority of page faults which resulted from changes in program locality occurred during relatively short time intervals.

COMPARISON WITH THE LRU AND WORKING SET REPLACEMENT ALGORITHMS

In order to compare different replacement algorithms, it is not enough to compare their measured efficiency, since every replacement algorithm will yield a high efficiency if the number of replacements is very small. This can be achieved by providing a large amount of memory. A better criterion for evaluating the performance of a replacement algorithm is the amount of main memory required to achieve a high efficiency level. For this reason the space-time product is a more valid measure for comparison.

Let us use the performance of a specific PFF Algorithm with $P=1/100$ page faults/msec to compare it with the LRU Replacement Algorithm for various numbers of allocated page frames. Figure 12 shows that the space-time products of the PFF Algorithm for all four programs are lower than the minimum space-time products achievable by the LRU Replacement Algorithm. This implies that the performance of the PFF Algorithm is better than that of the best LRU Algorithm.

The above results reveal the inefficiency involved in the fixed-size memory allocation. In such systems good performance can only be achieved if we allocate a number of page frames which is close to the optimal number that minimizes the space-time product. But even if we knew this optimal number, the performance of the PFF Algorithm would still be superior. This is especially true in those cases where the memory requirements vary drastically during execution (e.g., META7). If the memory requirements are relatively constant (e.g., FORTCOMP), then the performance of the LRU Algorithm is similar to that of the PFF Algorithm, provided that the optimal number of page frames is known a priori.

Figure 13 shows a comparison of the performance of the Working Set and PFF Algorithms. The space-time product is plotted for the working-set parameter τ and PFF-parameter P. For large values of τ, the space-time product of the Working Set Algorithm is usually lower than the space-time product of the PFF Algorithm for

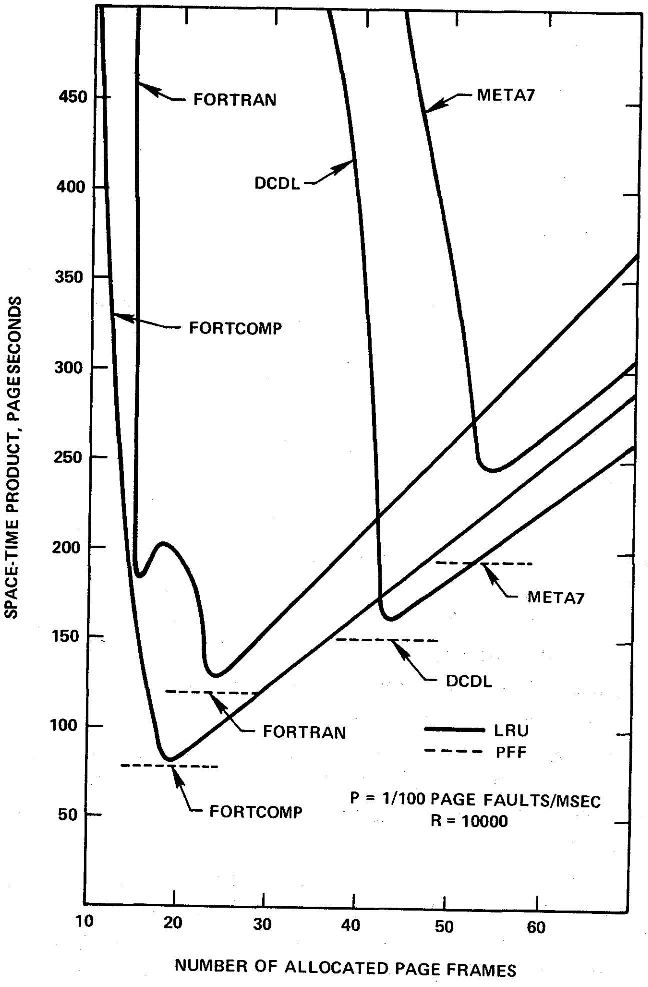


Figure 12—Performance comparison between the LRU and PFF algorithms

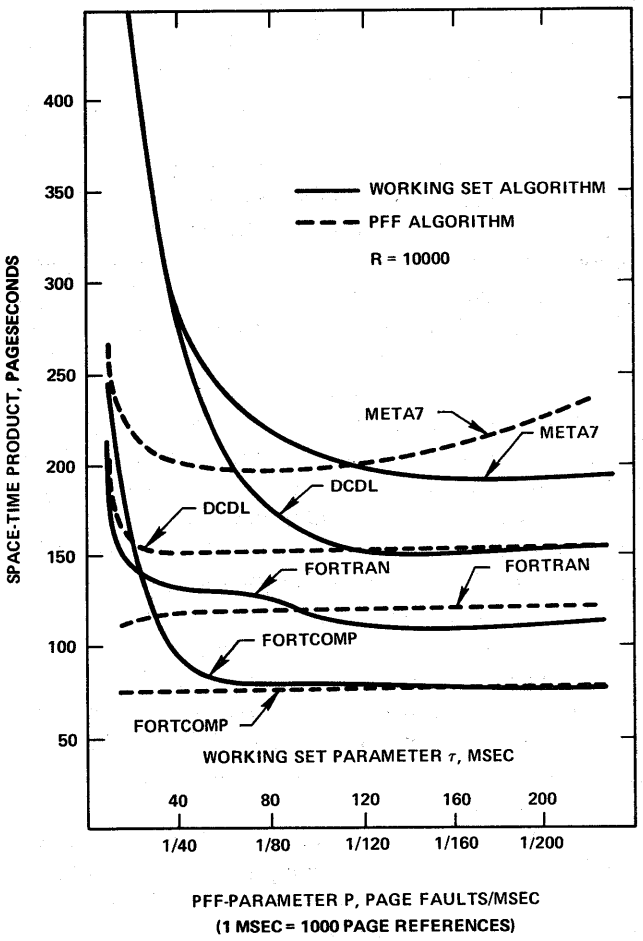


Figure 13—Performance comparison between the working set and PFF algorithms

corresponding values of P. Within the range of P and τ of interest, the space-time product of the PFF Algorithm is less sensitive to P than the space-time product of the Working Set Algorithm is to τ. A similar statement can be made with respect to the normalized efficiency and the average inter-page-fault-time (see Figures 4, 8 and 5, 9). Since the space-time product of the PFF Algorithm is relatively insensitive to the PFF-parameter P, we do not have to know an "optimal PFF-parameter" to come close to optimal performance. Further, the minimum space-time product of the Working Set Algorithm is comparable to that of the PFF Algorithm. The normalized efficiency and the average inter-page-fault-time of the PFF Algorithm are greater than the normalized efficiency and average inter-page-fault-time of the Working Set Algorithm for all corresponding values of P and τ. This shows that the performance of the PFF Algorithm is comparable to the performance of the Working Set Algorithm.

APPLICATION OF THE PFF ALGORITHM IN A MULTIPROGRAMMING ENVIRONMENT

Variable-sized memory allocation algorithms such as the Working Set Algorithm and the PFF Algorithm are useful in a multiprogramming environment where the main memory is shared by several programs. In this case the total amount of main memory not occupied by the resident supervisor can be considered as a pool of available page frames. These page frames are allocated to processes and returned to the pool according to the dynamically changing memory requirements of each individual process.

In the previous sections, we have used simulation techniques to study the performance of different replacement algorithms. It has always been assumed that there is enough main memory available so that a process can extend its memory space whenever needed. Any implementation of these variable-sized replacement algorithms, of course, must consider the possibility that the pool of available page frames is empty. The probability of this event is determined by the degree of multiprogramming and the type and size of the programs currently in the main memory. However, these variables can, to a large extent, be controlled by the process scheduling mechanism. Therefore, memory management is closely related to process scheduling in multiprogramming and time-sharing systems.

In order to reduce CPU idle time due to excessive page swapping, each process must be provided with enough main memory to keep the page fault frequency low. In a multiprogramming environment, it is crucial that this is accomplished without waste of memory space. The PFF Replacement Algorithm provides the supervisor with a means of achieving this effect which assures the same high efficiency level for programs of completely different types and sizes. In addition, the decrease policy of the PFF Algorithm continually tries to free those page frames which are no longer used by the process which enable processes to be run efficiently without wasting memory space.

The PFF Algorithm can also be very helpful for process scheduling since it gives the supervisor information about the required number of page frames for each process during execution. Once a process is removed from the main memory this information can be used to schedule this process for the next time quantum. In general, a process will be put on the processor queue only if there are enough available page frames in the pool. The information about the memory space of each process can also be used to decide which process has to be removed from the main memory if the page frame pool becomes empty.

There are many ways for the supervisor to make use of the information about program behavior provided by the PFF Algorithm. Further investigations might yield other interesting applications.

IMPLEMENTATION OF THE PFF REPLACEMENT ALGORITHM

The implementation of the PFF Algorithm is very simple. We need only a clock in the CPU to measure the time between page faults of every process. This clock measures the process (or virtual) time of each process. The current process time is recorded in the process' stateword. The page table entry can be used to determine which pages are residing in the main memory. For those paging systems that have a USE-BIT feature, this feature can be used to determine those pages which have been referenced during the time interval since the last page fault occurred. Whenever a page fault occurs, the USE-BITs are reset and the supervisor determines whether the process is operating below the critical page fault frequency level P. For this purpose the time of the last page fault has to be stored. If the last page fault occurred more than $T=1/P$ msec ago, the USE-BITs are used to determine which pages have to be removed from the main memory.

Let us now consider the overhead of the above mentioned operations. We know that:

1. The overhead is proportional to the number of page faults. Since the PFF Algorithm assures a low page fault frequency, the overhead is very low.
2. Due to sudden changes of program localities the virtual processing time between page faults is very short in many cases (see Figure 11). Whenever the time between page faults is less than $T=1/P$, no page frames are freed and therefore there is no overhead involved in the "decrease decision" in these cases.

From the above implementation discussion we know that the PFF Algorithm is much easier to implement, and requires less overhead to operate than both the LRU and Working Set Algorithms.

SUMMARY

A new type of replacement algorithm based on page fault frequency (PFF) is developed in this paper. This PFF Replacement Algorithm allocates memory according to the dynamically changing memory requirements of each process. It does not require prior knowledge of program behavior and can be applied to programs of different types and sizes. The PFF Algorithm uses the measured page fault frequency as the basic parameter for the memory allocation decision process. A high page fault frequency is considered to be an indication that a process needs more memory space to run efficiently. Thus whenever a page fault occurs the amount of allocated memory is increased if the page fault frequency lies above a given critical level P. P is called the PFF-parameter. The number of allocated page frames may be decreased if the page of allocated page frames may be decreased if the page fault frequency falls below this level P. In this case only those page frames are freed which have not been accessed between successive page faults. The PFF Replacement Algorithm adapts to dynamic changes in program behavior during execution. As a result, this algorithm is largely independent of individual program behavior and input data.

Measurement results from simulation of the PFF Algorithm for four different programs reveal that the performance (in terms of space-time product) of this algorithm is better than the performance of the best LRU Replacement Algorithm (for which the optimal memory space is known a priori), and is comparable to the Working Set Replacement Algorithm. Further, the performance is relatively insensitive to changes in the PFF-parameter P.

The implementation of the PFF Replacement Algorithm is simple and less complicated than that of LRU and far less complicated than that of the Working Set Replacement Algorithm. It does not require any additional hardware. Using the PFF Algorithm in a multiprogramming environment, the supervisor has control over the efficiency and memory requirements of all processes. Based on this information, the supervisor can perform efficient process scheduling and memory allocation. From this study, we conclude that the PFF Replacement Algorithm should have high potential for use in future virtual memory and multiprogramming systems.

REFERENCES

1 R L MATTSON J GECSEI D R SLUTZ L TRAIGER
Evaluation techniques for storage hierarchies
IBM Systems Journal 9 2 pp 78-117 1970

2 E G COFFMAN T A RYAN
A study of storage partitioning using a mathematical model of locality
Communications of the ACM 15 3 pp 185-190 March 1972

3 P J DENNING
The working-set model for program behavior
Communications of the ACM 11 5 pp 323-333 May 1968

4 W W CHU N OLIVER H OPDERBECK
Measurement data on the working set replacement algorithm

and their applications
Proceedings of the MRI International Symposium XXII
Polytechnic Institute of Brooklyn April 1972

5 G H FINE C W JACKSON P V McISAAC
Dynamic program behavior under paging
Proceedings of the 21st National Conference of the ACM pp 223-228 1966

6 L A BELADY
A study of replacement algorithms for virtual storage computers
IBM Systems Journal 5 2 pp 78-101 1966

7 E G COFFMAN L C VARIAN
Further experimental data on the behavior of programs in a paging environment
Communications of the ACM 11 7 pp 471-474 July 1968

8 M JOSEPH
An analysis of paging and program behavior
Computer Journal 13 1 pp 48-54 February 1970

9 L A BELADY C J KUEHNER
Dynamic space sharing in computer systems
Communications of the ACM 12 5 pp 282-288 May 1969

Experiments with program locality*

by JEFFREY R. SPIRN** and PETER J. DENNING***

Princeton University
Princeton, New Jersey

INTRODUCTION

For many years, there has been interest in "program locality" as a phenomenon to be considered in storage allocation. This notion arises from the empirical observation that it is possible to run a program efficiently with only some fraction of its total instruction and data code in main storage at any given time. That virtual memory systems can be made to run at all demonstrates that program locality can be used to advantage; and though it is certainly possible to write a program which violates the principles of locality, it seems one must go out of one's way to do so.

If a program is favoring a subset of its information at some particular time, we should very much like to know the identity of that subset. The set of favored pages† of information at a given time will be called the *locality* at that time. Using this information, we may answer such questions as "What behavior can be expected of the program in the near future?" or "How much storage should be allocated to the program at this time?" For some classes of programs the best we can do is estimate this locality, whereas for others we may be able to measure it exactly. The utility of this measurement is demonstrated by the fact that, for several models of program behavior, the policy "keep the current locality in memory" can be proved to be an optimal memory management policy. These models include the independent reference model,[1] the locality model,[2,3] and the least-recently-used (LRU) stack model.[4,5,6] For other locality processes, this policy appears to be nearly optimal.[3,11]

The means of measuring the locality, and the accuracy of the measurement, depend on one's definition of "locality." The definitions that have appeared so far in the literature can be classified into two categories: the *intrinsic* locality models, and the *extrinsic* ones.

Intrinsic models for locality assume that memory references emit from a program according to some (abstract) structure internal to the program itself. The locality in effect at a given time is a function of the internal state of the program at that time. Since the state of the program may not be known, it is usually not possible uniquely to determine the locality by examining the memory reference sequence of the program. Some examples of this type of locality model are page reference distribution functions,[7,8] the independent reference model,[1] the locality model,[2,3] and the LRU stack model.[4,5] Another example can be found in Reference 6, where, for $p>0$, it is assumed that there exists a sequence of sets of pages $W_p(1)$, $W_p(2), \ldots,$ $W_p(t), \ldots,$ such that $W_p(t)$ is the smallest set of pages containing the reference at time t with probability at least p.

Extrinsic models do not rely on any assumptions of internal program state. They define locality in terms of observable properties of the memory reference sequence of the program. Three examples of extrinsic locality are: (1) Given a sequence of time intervals, the "locality sequence" $L_1L_2 \ldots L_i \ldots$ is defined so that L_i is the set of pages referenced in the ith interval; (2) Given an integer $k \geq 1$, define a sequence of time intervals so that each locality L_i in the locality sequence $L_1L_2 \ldots L_i \ldots$ contains exactly k pages—i.e., exactly k distinct pages are referenced in the ith interval; and (3) A "working set" $W(t, T)$ is defined to be the set of distinct pages referenced among the last T references, and is a measure of the locality at time t.[9,10,11]

Intrinsic models are useful primarily for analysis and simulation. They are limited by the accuracy to which

* Work reported herein was supported in part by NSF Grant GJ-30126 and NASA Grant NGR-31-001-170.

** Present address: Division of Engineering, Brown University, Providence, Rhode Island 02912.

*** Present address: Department Computer Sciences, Purdue University, Lafayette, Indiana 47906

† We assume pages are all of the same size, containing at least one word each. Most of our results extend in a straightforward manner to systems in which the block size is variable, so that the assumption of paging is mostly a matter of convenience.

they simulate real programs. Due to the practical difficulty of measuring or estimating the locality, they may have little use in storage allocation. Extrinsic models are evidently more practical, since they define a measurement procedure; yet they are obviously limited by the extent to which the measurement taken reflects what the program is really doing. Such models are less suited for use in modeling, but they can be used conveniently to allocate memory.

Although there are many models for defining the concept of locality, little experimental verification of their accuracy has been undertaken. The working set model is perhaps the only exception.[12,13,14] Unless a given model can be shown to approximate closely the behavior of real programs, any analytic results obtained using the model are only of theoretical or academic interest. Accordingly, we have chosen in this paper to emphasize experiments which *test the ability of extrinsic measurements to estimate current intrinsic localities and predict future (intrinsic) localities*, and the ability of intrinsic models to simulate real world behavior.

Let us summarize the terminology that we shall be using for the various meanings of locality. If, as discussed above, a program's memory reference string is divided into (not necessarily equal) time intervals, the (extrinsic) *observed locality* L_i is defined to be the set of pages referenced in the ith interval. Since it may be difficult to determine the internal state of a program according to an intrinsic model, we usually in practice use the observed locality in the immediate past (such as the working set $W(t, T)$) as an estimate for the current intrinsic locality; this use is termed an *estimated* locality. If we assume something about the program's internal structure, we may be able to predict, on the basis of the current (estimated) locality, the most likely references in a future interval; this is termed the (intrinsic) *predicted locality.*

For some intrinsic models, the estimated locality can quite accurately (or even perfectly) determine the current intrinsic locality. Such models are clearly of special interest, and we shall discuss two of them. A third, the independent reference model, is in general not as well measured by the working set, but is presented for comparison.

Throughout this paper, it will be assumed that demand paging is being used and that a paging algorithm is optimal if it minimizes the expected probability of a page-fault in a given size memory.

MODELS FOR INTRINSIC LOCALITY

Consider an n-page program whose pages constitute the set $N=\{1, 2, \ldots, n\}$. A *reference string* $r_1r_2 \ldots r_t \ldots$ is the sequence of members of N generated by the program for given input data, where reference r_t is the number of the page containing the address referenced at time t (time being measured in terms of the number of memory references made by the program). Suppose a reference string has been divided up into intervals, and L_i is the observed locality in the ith interval. With respect to the given sequence of intervals, the reference string is considered to satisfy the *properties of locality* if:[10]

1. For almost all i, L_i is a proper subset of N;
2. For almost all i, L_i and L_{i+1} tend to have many pages in common; and
3. The observed localities L_i and L_{i+j} tend to become uncorrelated as j becomes large.

A program reference string is considered to have a high degree of locality if L_i is a small subset of N (statement 1), L_i and L_{i+1} differ by at most one page (statement 2), and the value of j for which L_i and L_{i+j} become uncorrelated is small compared to the length of the reference string.

A very general model for locality, displaying properties 1-3 intrinsically has been defined in Reference 3. It defines a sequence

$$(L_1, t_1)(L_2, t_2) \ldots (L_i, t_i) \ldots \qquad (1)$$

in which L_i is the ith intrinsic locality and t_i the *holding time* in L_i; the L_i are members of a specified set $\mathcal{L}$ of localities associated with the program, and are subsets of N. During its stay in L_i, the program generates some sequence of references $r_{i1}r_{i2} \ldots r_{it_i}$, over the pages of L_i only. The mechanism for generating the references from L_i is unspecified and may be arbitrary. The *current locality* L_t at time t is that L_i for which $t_1+\cdots+t_{i-1}<t\leq t_1+\cdots+t_i$. A probability structure can be imposed by specifying a transition matrix $[p(L, L')]$ among localities L and L' of $\mathcal{L}$, and a set of holding time distributions $h_L(t)$ for each L of $\mathcal{L}$

In the following sections we shall discuss some special cases of this general model. These cases are of practical interest to the extent that our experiments indicate agreement between localities predicted by these cases and the localities actually observed by using the working set model.

The very simple locality model (VSLM)

This model assumes a *fixed size* locality—i.e., the localities L_i in (1) are all of the same size l, where $1\leq l<n$. At any given time t, the probability of referencing an *interior page* (a member of L_t) is $1-\lambda$, and

the probability of referencing an *exterior page* (one not in L_t) and making a transition is λ. All l interior pages are referenced independently and with equal probability $(1/l)$. All $n-l$ exterior pages are referenced independently and with equal probability $(1/(n-l))$. When an interior page is referenced at time $t+1$, no change in locality occurs—i.e., $L_{t+1}=L_t$. When an exterior page is referenced, a change in locality occurs, but to a *demand-paging neighbor* only—i.e., $L_{t+1}=L_t+r_{t+1}-y$ where y is chosen at random from L_t. The unconditional probability of referencing an interior page is at least as large as that of referencing an exterior page, i.e.,

$$\frac{(1-\lambda)}{l} \geq \frac{\lambda}{n-l}, \qquad (2)$$

which is equivalent to the condition $\lambda \leq (n-l)/n$. This model has two important parameters—the locality size l and the transition probability λ—and will sometimes be called the *two-parameter model.* Note that the mean holding time in a locality is $1/\lambda$. For this model, the storage allocation rule "keep the current locality in memory" has been proved optimal.[3]

It can easily be shown that for programs which fit this model, it is impossible to determine absolutely the current intrinsic locality from observations on the generated reference string. We shall consider next the accuracy with which we can estimate the locality by an extrinsic model, namely, the working set.

As mentioned, the working set $W(t, T)$ is the set of distinct pages referenced among the references $r_{t-T+1} \ldots r_t$. If we desire to use the working set to estimate the locality, we must specify T, the window size. The choice of T must satisfy two criteria: (1) it must be large enough so that all pages within the locality are referenced with high probability, and (2) it must be small enough so that the likelihood of more than one locality transition within the window is low (for several transitions would introduce error). Although it is not obvious that a suitable T can be found, it is the case that for reasonable parameters of the VSLM, not only does a T exist, but its value is not especially critical. For the VSLM, condition 2 will hold whenever $T \leq 1/\lambda$, and our experiments verify that such values of T typically exist.

We shall consider the working set to be a good estimate of a VSLM locality when two criteria are satisfied: (a) the average working set size is approximately equal to l, the locality size, and (b) the average missing-page probability when the working set is kept in memory is approximately λ, the probability of referencing outside the locality. Plots of working set sizes and missing-page probabilities for various values of n, l, and λ show that,[3] for small λ (.01 or less), a value of T on the order of 5 or 10 times the locality size will do an excellent job of achieving criteria (a) and (b) above, irrespective of n and l. Furthermore, for small values of λ, the values of the working set size and missing-page probability level off and are nearly constant in a large neighborhood of T, indicating that the choice of T is not too critical for these values of λ.

For large values of λ (in excess of 0.05), the working set apparently does not provide as good an estimate of the locality. In this case, the working set size and missing page probabilities do not tend to level off at the values of l and λ, respectively. Furthermore, the value of window size needed to get the missing-page probability equal to λ gives a working set size as much as 20 percent too large.

The simple LRU stack model (SLRUM)

This model is based on the memory contention stack generated by the LRU (least-recently-used) page replacement algorithm.[5] This stack is simply a priority ordering on all pages of a program according to the time of their most recent usage. Thus, the first position (top) of the stack is the current reference, the second position is the next most recently used page, and so on. When the page in stack position i is referenced, it is moved to the top, and all the pages which were in positions $1 \ldots i-1$ are pushed down one position. Specifically, if $\mathbf{s}(t) = (x_1, \ldots, x_n)$ is the stack at time t and the page at position i is referenced, the stack at time $t+1$ is $\mathbf{s}(t+1) = (x_i, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$.

To create the simple LRU Stack Model, we assign to each position of the stack a fixed, independent probability. We will denote these probabilities $a_1, \ldots, a_n$, where n is the number of pages in the program (and thus the number of stack positions) and $a_1 + \cdots + a_n = 1$. The a_i are termed *stack distance probabilities* with i being the distance (from the top of the stack). At any given time stack position i will be chosen with probability a_i; if it is chosen, the page in that position becomes the current reference and is brought to the top of the stack, as above.

If we make suitable restrictions on the a_i, we can cause this model to exhibit locality. In Reference 3, the requirement is made that the a_i be monotonically non-increasing as one goes down the stack $(a_1 \geq \cdots \geq a_n)$*. If, under this restriction the stack is divided at any point, the pages in the stack positions above the division are all more probable than those below the division. Specifically, if the stack at time t is $\mathbf{s}(t) = (x_1, \ldots, x_n)$, we can define a locality of size l

*This requirement can be weakended slightly to $\min\{a_1 \ldots a_m\} \geq \max\{a_{m+1}, \ldots, a_n\}$ for LRU paging in a memory of size m [3].

(for any l, $1 \leq l < n$) to be the pages $\{x_1, \ldots, x_l\}$. By dividing the stack at successive distances, a hierarchy of localities may be defined. This hierarchy represents a full ordering of localities, in that any given locality contains all of those smaller than it.

Note that the SLRUM is a slight generalization of the VSLM. At any given time, the probability of a locality transition is

$$\lambda = a_{l+1} + \cdots + a_n$$

since a transition occurs if and only if the distance exceeds l. Moreover, when a new locality is entered, it is the demand paging neighbor of the former locality.

It can be shown that, if m_t is the amount of memory allocated at time t, the optimal storage allocation rule for reference strings generated by this model is: "keep the top m_t elements of the stack $\mathbf{s}(t)$ in memory, for all t."[3] It follows in particular that, if m_t is the size of the working-set $W(t, T)$, the working-set policy is optimal for this model (note that $W(t, T)$ then contains precisely the top m_t elements of $\mathbf{s}(t)$). If m_t is fixed, it follows that the LRU paging algorithm is optimal for this model.

It is worth re-emphasizing that the working set $W(t, T)$ and the observed LRU stack (i.e., the one maintained by the LRU paging algorithm) both measure *exactly* the locality according to the SLRUM. For this intrinsic model, therefore, extrinsic measures provide an exact measure of locality.

The independent reference model (IRM)

In this model, the page references $r_1 r_2 \ldots r_t \ldots$ are assumed to be independent trials under some fixed probability distribution $\{c_1, \ldots, c_n\}$. In other words, the probability of referencing page i at time t is given by the stationary probability

$$\Pr[r_t = i] = c_i \qquad (3)$$

Note that consecutive page references are taken according to these probabilities without regard to the previous references made by the model.

We may form a priority list for this model by ranking the pages according to decreasing probability—i.e., there is a *fixed* priority list $(1, 2, \ldots, n)$ where $c_1 \geq c_2 \geq \cdots \geq c_n$. Given a value of l, define a locality of this model to consist of pages $L(l) = \{1, 2, \ldots, l-1\}$ and that page x which was most recently fetched into memory (note that $l \leq x \leq n$), so that a locality is of the form $L(l) + x$. The rule "keep the pages of $L(m)$ in memory at all times," for memory size m pages, is known to be optimal for the IRM.[1] As in the VSLM and the SLRUM, transitions occur between demand-paging neighbors only. Unlike these other two models, however, the transition probability varies in time, being $c_l + \cdots + c_n - c_x$ whenever the locality is $L(l) + x$. The important difference between the IRM and the two previous models is, the localities of the IRM are essentially static in content whereas those of the VSLM and the SLRUM are changing in content. We shall see that, because of this difference, the IRM produces poor fits to actual programs.

CRITERIA FOR EXPERIMENTATION

It is obvious that if one were to try to correlate the reference strings produced by a model with the observed reference string of some given program, one would have very low success: Direct correlation is much too stringent a requirement to place on a model. A more reasonable, though indirect, way is to correlate interreference densities; that is, the time between consecutive references to the same page. However, even this method is likely to be inconclusive (at least over relatively short reference strings), since experimentation shows that the interreference densities of real programs tend to be quite irregular in shape, many zeros being interspersed between non-zero probabilities.

We were interested primarily in testing whether or not the reference strings generated by a given model induce the same paging behavior as those generated by a real program. Thus, we did not care about fitting strings of references within a locality, since these will be transparent to the paging system, assuming the locality is retained in memory. We were concerned, however, with locality transition behavior. Moreover, since we wanted to use the working set to estimate the locality, we desired the model to have similar working set behavior, at least on the average.

Taking these factors into account, we decided to try to fit two types of curves. The first is the *average working set size* $w(T)$, which gives the average working set size in an interval as a function of the window size T. The other is the *average missing-page probability* $q(T)$ as a function of the window size T, when exactly the working set is kept in memory at all times. It has been proved that the latter curve is (essentially) the derivative of the former,[10] and thus we are in fact fitting the model to the working set size curve and its derivative.

Now, consider the probability of a given page's not being in the working set under window size T: this can be shown equivalent to the probability of the interreference interval for the given page being greater than T.[10] Thus, the missing page probability $q(T)$ corresponds to the complementary cumulative overall interreference probability distribution. In this way, a close fit by a model to the observed missing page probability curve guarantees a close fit to the observed overall interreference *distribution*, even though the fit to the observed *density* will, as commented earlier, tend to be quite poor.

Of course, this method of model fitting has its disadvantages. Its primary limitation is that both the curves $w(T)$ and $q(T)$ are averages measured over an interval. If the interval is too large, any non-stationary behavior will tend to be masked on the average. For this reason, most measurements were taken over what we consider to be a suitably small interval (short compared to the lifetime of the program), in most cases 20K references (about 10,000 instructions). This necessitated taking several measurements in various parts of a given program's reference string. For comparison, measurements over a larger interval, 300K references long, were also taken.

We decided for these experiments to ignore the distinction between instruction and data references. Modern computers tend to make great use of such operations as register-to-register instructions, indirect references, and multiple data reference instructions (such as LM and STM on the 360). The more a program makes use of these operations, the less true the tendency for instruction and data references to alternate. We decided not to make detailed studies of how instruction and data references are in fact mixed in practical reference strings, as this question was secondary to our interest in locality behavior. Moreover, most modern systems do not make any serious attempts to distinguish "instruction working sets" from "data working sets" in their storage allocation procedures. Nonetheless, the effects of such a distinction may be significant, and constitute a worthwhile project for future research.

For each reference string segment tested, the observed (OBS) working set curve, independent reference probabilities $\{c_i\}$, and LRU stack distance probabilities $\{a_i\}$ were measured. A single-pass algorithm for measuring the working set curve is given in Reference 10. The independent reference and LRU stack probabilities were determined by counting references to each page and to each stack position, respectively. The value used for n, the size of the total program's page set, was the total number of distinct pages actually referenced in the reference string segment being studied. Pages which were never referenced in the interval of measurement were not counted in the value of n. This was done mostly for convenience, and should have little or no effect on the results.

Also included for comparison was an attempt to fit the working set curve to the following exponential function

$$w(T)=n(1-e^{-BT}) \quad B>0$$

where B is a parameter. This will be termed the *exponential model* (EXP).

Using the measured values for the independent reference and LRU stack probabilities, the working set curves for these two models were computed. (Algorithms for computing the working set curves of the various models are given in Reference 3.) For the locality model, the parameters l and λ were chosen to give the lowest mean-squared relative error for the set of window sizes 10, 20, 30, . . . , 1000, against the observed $w(T)$ curve. The same procedure was repeated for the exponential model to determine a value of the parameter B.

DESCRIPTION OF RESULTS

Programs on two machines were tested for fits with the various models. The PAL assembler on the Digital

CHART 1—Description of Programs Measured

Ref. Str. No.	Machine	Page Size (words)	Description	Refs. Skipped	Refs. Measured
0	PDP-8	128	Assembler, Pass 1	0	20K
1	PDP-8	128	Assembler, Pass 1	100K	20K
2	360	256	FORTRAN (G) COMPILER	1K	20K
3	360	256	FORTRAN (G) COMPILER	200K	20K
4	360	256	Small FORTRAN job. One main loop.	1K	20K
5	360	256	Small FORTRAN job. One main loop.	100K	20K
6	360	256	Small FORTRAN job. One main loop.	1K	300K
7	360	256	Medium FORTRAN job. Several Subroutines.	1K	20K
8	360	256	Medium FORTRAN job. Several Subroutines.	100K	20K
9	360	256	Medium FORTRAN job. Several Subroutines.	1K	300K

CHART 2—Values of Parameters

Ref. Str. No.	Total pages refd.	VSLM l	VSLM λ	EXP B
0	11	4	.0025	.0013
1	12	4	.0027	.0015
2	35	5	.014	.00080
3	38	7	.022	.0012
4	20	3	.030	.0025
5	20	4	.020	.0020
6	20	4	.021	.0021
7	22	4	.024	.0020
8	20	3	.029	.0024
9	31	4	.014	.00085

Equipment PDP-8 was run using a page size of 128 words, the standard page size for the machine. Several IBM 360 programs were run, including two FORTRAN jobs and the FORTRAN (G level) compiler itself. The 360 page size was chosen arbitrarily to be 256 words. Chart 1 gives a description of each program. The "reference string number" refers to a reference string segment from each program. In particular, we expressed the reference string in the form $r_1r_2 \ldots r_kr_{k+1} \ldots r_{k+x} \cdots$, where k is the number of "references skipped" and x the number of "references measured." In other words, $r_{k+1} \ldots r_{k+x}$ is the reference string segment over which we attempted to fit the models.

Curve fit results

Chart 2 gives the values of various measured or best-fit parameters. It is important to note that the best-fit VSLM locality size l was typically under 20 percent of the program size n, that the locality transition probability λ was typically in the range 0.01 to 0.03, and that the locality transition time $1/\lambda$ was typically in the range 30 to 100 references. Reference strings 0 and 1 were exceptions, having much lower transition probability λ than the others, this being due undoubtedly to the severely limited amount of memory on the PDP-8 (4K words).

Chart 3 gives the results of the working set curve

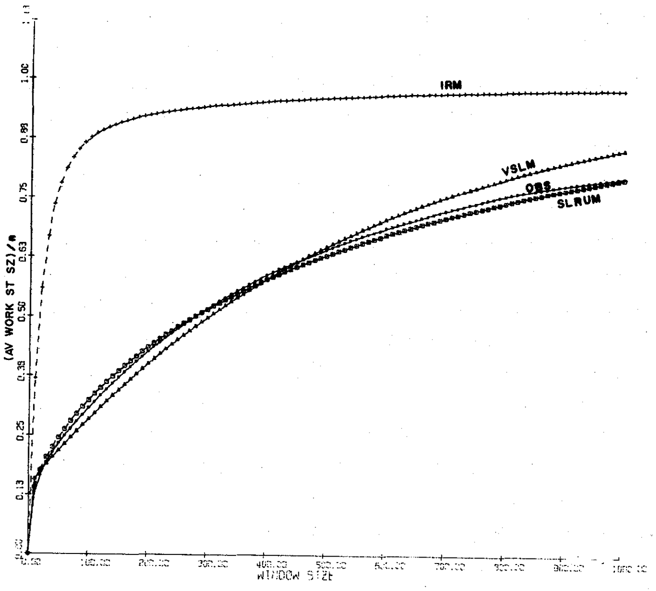


Figure 2—Working set size (Ref. St. 4)

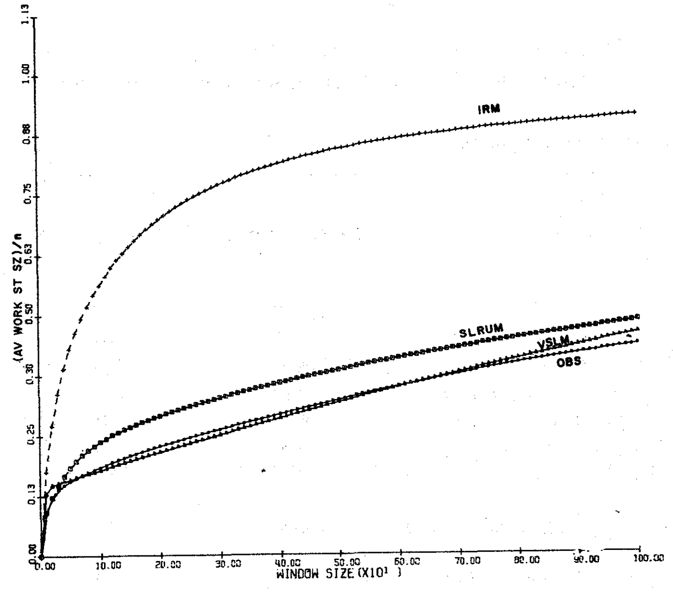


Figure 1—Working set size (Ref. St. 2)

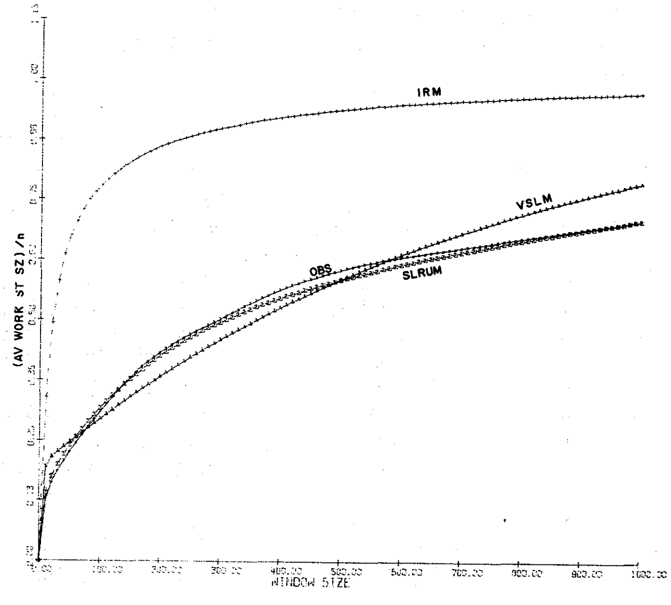


Figure 3—Working set size (Ref. St. 6)

CHART 3—Fits to mean working set size curve

Ref. Str.	VSLM		EXP		IRM		SLRUM	
No.	avg. error	worst error	avg. error	worst error	avg. error	worst error	avg. error	worst error
0	7.5%	56%	37%	99%	84%	97%	28 %	33 %
1	6.2	49	38	99	97	106	19	24
2	6.0	49	32	97	161	208	20	29
3	11	58	32	96	109	146	6.5	8.2
4	5.3	30	20	95	95	246	2.6	7.7
5	10	53	26	96	77	200	2.3	7.9
6	9.9	54	25	96	86	210	2.6	8.1
7	8.0	51	24	96	98	225	2.8	7.9
8	6.5	29	22	95	85	207	2.9	8.9
9	10	56	31	97	162	291	8.3	9.4

fittings for the various models, and Chart 4 gives the corresponding results for the missing page probability curve. Two error measures are listed for each fit: "average relative error" over the curve, and the "worst case relative error." Except for the IRM, the worst errors occurred for very small values of T (less than 10); for the IRM, the worst errors occurred for the largest values of T (above 500). All errors are shown as per cent of the observed value. The "average relative error" is only an approximate value: it is found by taking the square root of the previously mentioned mean squared relative error (it can be shown that this represents an upper bound to the true average of the absolute values of the errors). The worst case error is the largest relative error considered over all integer window sizes in the range 1 to 1000.

It seems apparent from the data that the SLRUM performs the best over all in approximating the two curves, with the VSLM a close second. The fits of these two models are usually very good on the working set curve. The errors in fitting the missing page probability curve are larger, even unacceptably large in some cases. However, it can at least be said that even for this curve, these two models perform much better than either of the others, again with the SLRUM slightly superior. We can conclude from this that the models are better for predicting a program's memory demands than for predicting its page-fault probability; further refinements to the models are required to achieve the latter goal.

Because of its static treatment of locality, the independent reference model is the worst model of the four. It consistently overestimates the working set size, usually by a factor of 2 or 3.

Figures 1-3 show typical working set curves, and Figures 4-6 show typical missing page probability curves. All six figures show the observed (measured) curve OBS, and the results of attempting to fit each model. (EXP was omitted to aid in readability). Figures 7-9 show typical stack distance probabilities; all such curves show that the monotonically non-increasing assumption of the a_i tends to be valid for the majority of values of i. (Note the logarithmic vertical axis on these figures).

CHART 4—Fits to Missing Page Probability Curve

Ref. Str.	VSLM		EXP		IRM		SLRUM	
No.	avg. error	worst error	avg. error	worst error	avg. error	worst error	avg. error	worst error
0	30%	228%	266%	407%	77%	376%	84 %	197%
1	36	190	301	413	167	419	85	181
2	29	132	93	127	133	426	27	118
3	82	131	157	207	70	210	5.1	26
4	32	94	40	90	103	479	16	48
5	85	157	119	224	74	417	19	43
6	72	153	100	192	81	435	18	38
7	37	158	62	110	93	450	9.5	37
8	30	92	48	91	89	420	10	37
9	57	190	117	195	102	609	14	40

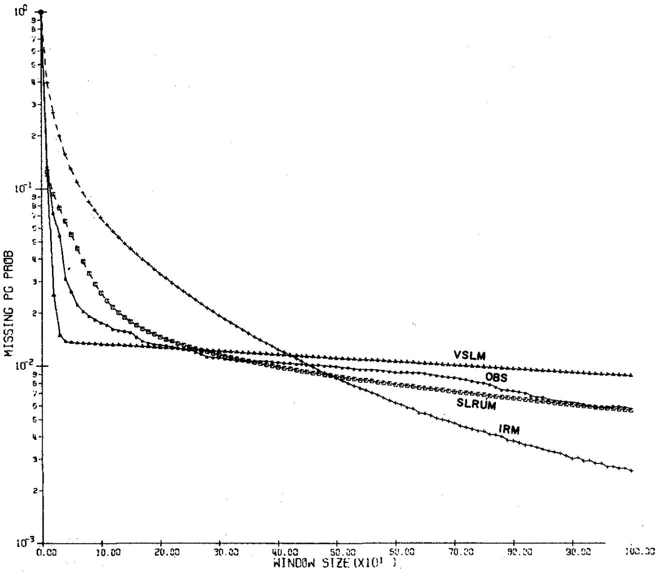


Figure 4—Missing page probability (Ref. St. 2)

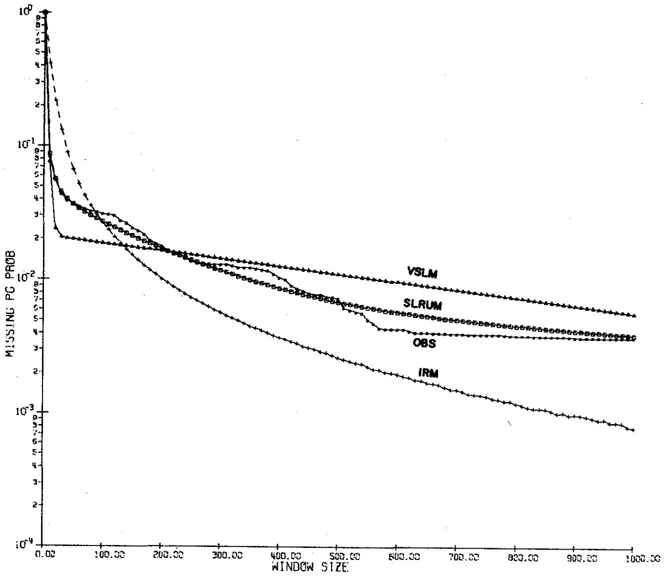


Figure 6—Missing page probability (Ref. St. 6)

Several other statistics of interest appear in Chart 5. q_I is the sum of the $n-l$ lowest measured independent reference probabilities; it gives an indication of the performance which could be expected if the program were in fact an IRM program allocated l pages of memory. q_w is the missing page probability for the working set with window size T_w, where T_w is chosen to make the average working set size equal to l. Thus, q_I and q_w apply to the same average memory size. Notice that q_I is typically an order of magnitude greater than q_w, showing much more dramatically how pronounced are the dynamic effects of locality: The assumption of static locality would have led us to predict missing page probabilities in the order of q_I whereas in fact they were in the order of q_w. This re-emphasizes the poor performance introduced by a model assuming a static locality.

It is also notable that in every case, $T_w < 1/\lambda$, where

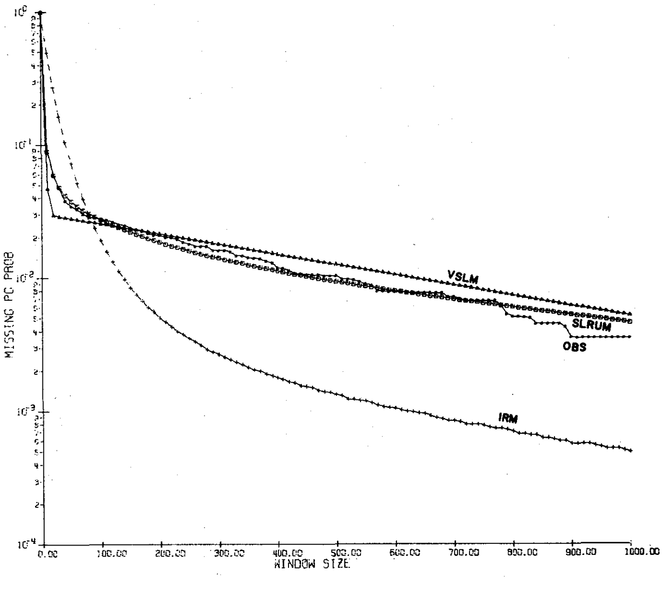


Figure 5—Missing page probability (Ref. St. 4)

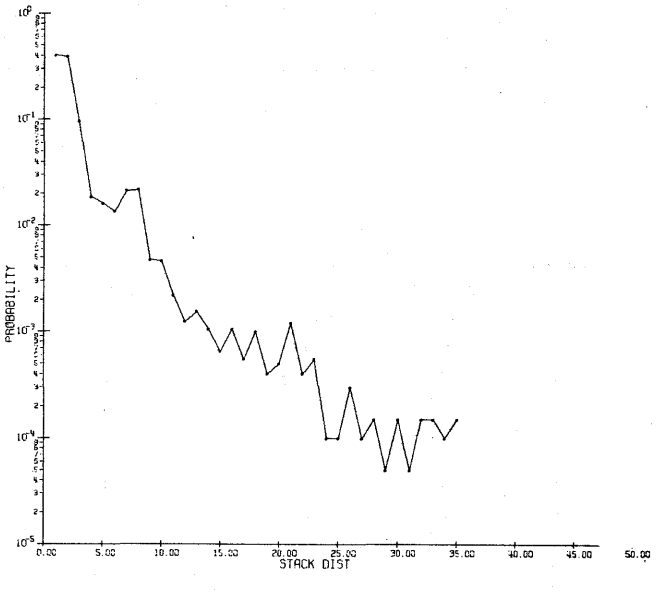


Figure 7—Distance distribution (Ref. St. 2)

1/λ is the expected interval between locality transitions in the VSLM. Thus, it is unlikely that more than one such transition will occur in this size window, so that the working set will be a good estimator of the VSLM locality for all tested programs.

EXTENSIONS TO THE *SLRUM*

Attempts have been made to improve the SLRUM by increasing the complexity of the process by which stack distances are generated. Shedler and Tung,[5] for example, analyze a stack with a Markov process substituted for the a_i. To our knowledge, no attempts have been made to validate any extensions to the SLRUM, other than that which we shall describe below.

CHART 5—Additional Statistics

Ref. Str. No.	q_I	q_W	T_W	π_S	h_S
0	.14	.0088	75	0.	—
1	.09	.024	45	0.	—
2	.33	.035	34	.038	10.6
3	.36	.071	40	.11	12.7
4	.69	.071	15	.015	5.5
5	.48	.042	36	.015	5.7
6	.49	.041	36	.015	5.7
7	.54	.046	31	.0084	4.2
8	.63	.070	15	.0040	4.0
9	.57	.030	43	.0056	4.0

A very simple attempt was made to improve the performance of the LRU stack model. It was imagined that stack distances would be selected, as before, according to the a_i, and the a_i would be biased toward short stack distances. Occasionally, however, a new set of probabilities, the b_i, would take effect for a short time; these would be biased toward long stack distances. The distribution $\{a_i\}$ corresponds to the intuitive concept of "drifting slowly among neighboring localities," whereas the distribution $\{b_i\}$ to the notion "jumping suddenly to very different localities," or "scrambling up the entire stack." The choice between $\{a_i\}$ and $\{b_i\}$ would be determined by a 2-state Markov chain.

As has been suggested earlier, there is a distance string $d_1d_2 \ldots d_t \ldots$ associated with the program's reference string $r_1r_2 \ldots r_t \ldots$ being measured. Given the distance string, our problem was to determine which distances should be considered data points for the $\{a_i\}$ distribution and which for the $\{b_i\}$ distribution. Somewhat arbitrarily, we decided to count the distances toward the $\{b_i\}$ distribution whenever the majority of the last four successive distances exceeded four (four was chosen since it represented a typical VSLM locality size); distances would continue to be counted toward the $\{b_i\}$-distribution until four successive distances were all at most four, in which case distances would be counted toward the $\{a_i\}$-distribution. The measured value for the steady state probability π_s of the $\{b_i\}$ (or "stack scrambling") state is shown in Chart 5; π_s is an indication of the fraction of time the program spent making large jumps between localities. Except

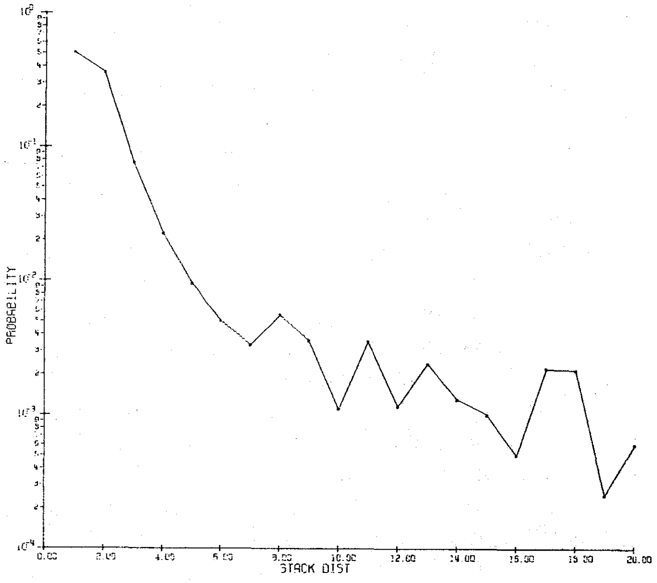


Figure 8—Distance distribution (Ref. St. 4)

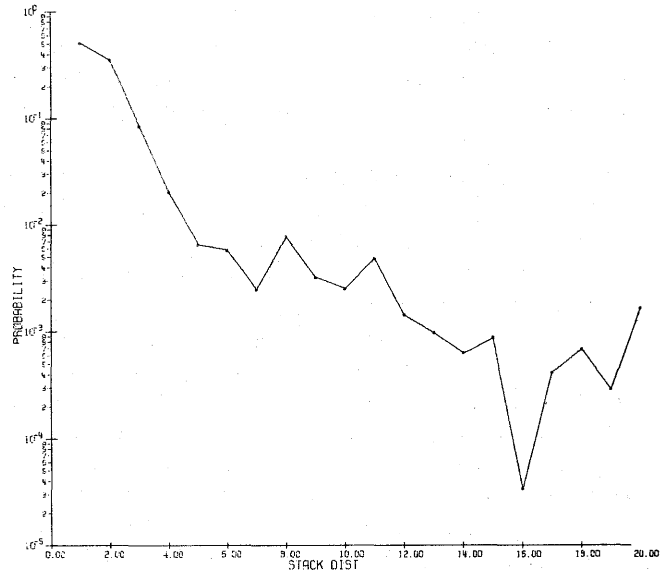


Figure 9—Distance distribution (Ref. St. 6)

for reference string 3, all the programs seemed to spend under 4 percent of their time jumping localities—i.e., they seemed to spend in excess of 96 percent of their time obeying the properties of locality.

Chart 5 also shows the mean holding time h_s in the $\{b_i\}$-state. In all cases, h_s was at least as large as the VSLM locality size l, suggesting that, when scrambling is over, the resulting locality is likely to be disjoint from the original locality.

As might be anticipated, however, the working set size and missing page probability curves generated by this extended model were in all cases indistinguishable from those produced by the SLRUM. This is because the transitions between the $\{a_i\}$ and the $\{b_i\}$-states occur independently of the process which generates stack distances. Apparently, it is necessary to make the stack-scrambling process correlated directly with the stack distance generating process, perhaps by generating distances directly from a Markov chain. Shedler and Tung's approach represents one possible solution,[5] though as yet unvalidated.

CONCLUSIONS

We have attempted here to validate experimentally several intrinsic models for the concept of program locality. We have done this with particular regard to the use of the working set as an estimator of the (intrinsic) locality. We have tried to take examples of both system software (a compiler and an assembler) and user programs, and have attempted to fit each of the models to the observed behavior of each given program.

Fitting was attempted to the measured working set size and missing page probability curves. In this way, reasonable approximations to the paging behavior of the actual programs could be obtained, without having to consider other details of the programs of less importance in paging.

Two models appear to produce good approximations to real world behavior: the two-parameter simple locality model and (especially) the LRU stack model. The independent reference model, because of its static concept of locality, does very poorly.

The working set is a good estimator of the simple two-parameter model's locality, provided the locality does not change too rapidly; we observed no case in which the locality was changing too rapidly for the working-set to be a good estimator. The working set *exactly* measures the locality in the case of the LRU stack model and is thus nearly optimal for programs whose behavior can be closely approximated by this model.

The principal conclusions to be drawn from this work are:

1. There exist non-trivial cases in which the working-set memory management policy is optimal, and evidence suggesting it will perform quite well when reference strings are generated by locality processes other than the ones studied here.
2. The concept of a "locality size" is not sharply defined, as in the case of the simple two-parameter model; it is instead a graduated concept, as in the LRU-stack model.
3. The locality at any given time receives the vast majority of references, is small compared to the program size, and is constantly changing in membership.
4. There is a tendency for transitions to occur between neighboring localities for the vast majority of the time, transitions among disjoint localities being relatively infrequent.

Stack models appear to hold great promise of being good models for program behavior, especially as we gain a better understanding of the processes by which stack distances are generated.

ACKNOWLEDGMENTS

We are grateful to J. J. Horning and K. Sevcik of the University of Toronto for many useful ideas and insights relating to intrinsic and extrinsic concepts of ocality.

REFERENCES

1 A V AHO P J DENNING J D ULLMAN
Principles of optimal page replacement
J ACM 18 1 January 1971 pp 80–93

2 P J DENNING J E SAVAGE J R SPIRN
Some thoughts about locality in program behavior
Proc Brooklyn Polytechnic Institute Symposium April 1972

3 ———
Models for locality in program behavior
Princeton University Department of Electrical Engineering Computer Science Technical Report TR-107 April 1972

4 P H ODEN G S SHEDLER
A model of memory contention in a paging machine
IBM Research Report RC-3053 September 1970

5 G S SHEDLER C TUNG
Locality in page reference strings
IBM Research Report RJ-932 October 1971

6 E G COFFMAN JR T A RYAN JR
A study of storage partitioning using a mathematical model of locality
Comm ACM 15 3 March 1972 pp 185–190

7 J E SHEMER G SHIPPEY
Statistical analysis of paged and segmented computer systems
IEEE Trans Comp EC-15 6 December 1966 pp 855–863

8 J E SHEMER S C CUPTA
On the design of Bayesian storage allocation algorithms for paging and segmentation
IEEE Trans Comp C-18 7 July 1969 pp 644–651

9 P J DENNING
The working set model for program behavior
Comm ACM 11 5 May 1968 pp 323–333

10 P J DENNING S C SCHWARTZ
Properties of the working set model
Comm ACM 15 3 March 1972 pp 191–198

11 P J DENNING
On modeling program behavior
Proc AFIPS Conf Vol 40 Spring Joint Computer Conference 1972

12 J RODRIGUEZ-ROSELL
Experimental data on how program behavior affects the choice of scheduler parameters
Proc 3rd ACM Symposium on Operating Systems Principles October 1971

13 W DOHERTY
Scheduling TSS/360 for responsiveness
Proc AFIPS Conf Vol 37 Fall Joint Computer Conference 1970 pp 97–112

14 W W CHU N OLIVER H OPDERBECK
Measurement data on the working set replacement algorithm and their applications
Proceeding of the Polytechnic Inst of Brooklyn Symposium on Computer-Communications and Teletraffic April 1972

TASSY—One approach to individualized test construction

by THOMAS L. BLASKOVICS and JAMES A. KUTSCH, JR.

West Virginia University
Morgantown, West Virginia

During the past ten years universities and the computing industry have seen the development of a new mode of teaching called Computer Aided—or Assisted—Instruction (C.A.I.). This new field, emerged as an attempt to meet and deal with the growing criticism and frustration of students, employers, legislators, and faculties, which stemmed from our inability to prepare students adequately.

Several very creative C.A.I. projects were directed toward providing a whole new system of instruction. However, to date, the success of C.A.I. has been limited, at best. PLANIT, PLATO, LYRIC, COPI, COURSEWRITER, and others have not been able to meet the needs of the teaching community. The problems reported by the major projects are only in part bounded by the technology of the computer.

At West Virginia University, we watched the development of these systems with great interest and concern, because we, like other universities, were faced with the same problems. We carefully examined several of the better-publicized systems with an eye toward implementing one of them to meet our instructional needs. As we analyzed the systems we discovered that C.A.I. systems:

1. Were too machine-dependent to allow a feasible implementation without scrapping our existing hardware (an IBM 360/75).
2. Were too expensive in terms of Core requirements.
3. Made (what appeared to be) unreasonable demands upon the instructor in terms of intimate knowledge of programming and/or computer technology.
4. Were "not yet available but would be soon" even though the projected date had slipped by several times.

We also found that the present C.A.I. systems were too monolithic. The decision to change a course or set of courses to a C.A.I. approach requires a "go-for-broke" commitment. We found that users did not like the non-incremental requirement of C.A.I. This was not too surprising to us when we considered that most of the end users who we were concerned about had little or no experience with computers.

Because of the many know factors in C.A.I. systems and the type of commitment required, many potential users we surveyed were reluctant to commit themselves and their resources to a C.A.I. effort.

As we were considering the major C.A.I. systems, we were also engaged in trying to determine what our students felt to be their own needs with regard to a college education. Along with study of student needs, we tried to discover the needs of the faculty with respect to their problems in teaching. The results of our study suggested that two problems existed. Students indicated that a major source of frustration (and possibly aggression) toward the University stemmed from a lack of feedback from the "establishment" regarding individual progress. A second, and only slightly less important, frustration was stated to be the lack of relevance of course material. Interestingly enough, the students felt that given the feedback, they would be able to deal with the problem of relevance themselves. Student evaluations of professors indicated that where amount of feedback was high, and good, relevance was not a problem. The faculty, by and large, agreed with the students feedback was a problem; however, they saw relevance as being more critical to the learning process. The faculty also indicated that they did not have an easy means to provide feedback.

In analyzing our findings, it appeared to us that one reason for the limited success of C.A.I. in other universities was possibly that it might have been the wrong solution for the problem facing the university at the present time.

One of the claims of C.A.I. is that it allows the instructor to individualize: to tailor the instructional experience to the student through a series of incremental feedback statements. This seemed to imply that the instructor would be spending more time with the student. Our observation of ongoing C.A.I. systems indicated that, once the horrendous task of programming had been accomplished, the instructor retreated to his office or lab either to write new programs, to collapse, or to become more deeply involved in his own research again. The net effect of C.A.I. was to make the student more dependent for help upon the machine, or if he were available, upon a graduate assistant. In some cases, the instructor developed a bad case of "Blinking Light Syndrome" and spent his time diddling with the machine.

Since C.A.I. seemed fraught with problems, we decided to look more closely at the problems of feedback. We made an assumption (tentatively) that the instructor was able to teach the material. Analysis of the instructor's time indicated that he spent a large portion of time developing, administering, and scoring exam questions, keeping track of what his teaching assistants were doing, worrying about the security of his files—and precious little time actually teaching. In most classes, the most feedback that students received were scores on one or two major exams and the final. In almost all cases we observed, the feedback was delayed until as much as two to three weeks following the respective tests. The students had virtually no opportunity to analyze their performance, or to learn where their deficiencies might be. It appeared that the feedback system we were currently using could not be seen, by any stretch of the imagination, as a learning experience. (In some cases it was viewed by the students simply as a means of satisfying the sado-masochistic tendencies of the faculty.)

In an analysis of one undergraduate course of 325 students, we found that the teaching assistants spent better than 10 percent of their time in purely mechanical activities, such as distributing, proctoring, and scoring tests. In addition, a course manager spent approximately 50 percent of her time doing clerical work necessary to keep the student's grades up-to-date, etc. Initially, we felt that the time commitments were rather high; until we realized that the assistants and course manager had almost total responsibility for 2,275 hours of testing in one semester.[1]

Further consideration of the function of efficient feedback suggested utility for both students and instructor. For the former, it provides (1) a test score, (2) a diagnostic evaluation of material learned (and/or not learned), and (3) (hopefully) some prescription to remediate his problem.

In 1971, Baker presented the state of the art in Computer Based Instructional Management Systems (CBIMS). He suggested that the instructor is not only a teacher, but also the manager of a rather complex system of activities designed to help the student learn something.[1]

He indicated that "a major facet of this managerial task is composed of the mechanical tasks of scoring homework, test papers, and keeping track of what instructional materials a student has used." Our discussions with faculty and teaching assistants tended to support the notions put forth by Baker.

Each of the systems Baker reviewed was designed to provide for the four major functions of any CBIM system, namely: test scoring, diagnosing, prescribing, and reporting. However, the actual operation of the systems seemed to be very awkward, and required that the student participate on some fixed schedule. Another difficulty that we observed was that the present CBIM systems seemed to double the work of the faculty member in that he had to develop essentially two sets of testing material—one set for the diagnostic function and another for the examination function.[2]

It appeared to us that if a system with interactive capabilities could be developed, it might resolve much of the awkwardness and restrictiveness we had observed. With regard to the second problem, having to maintain double sets of items, we asked the faculty why not let the students use the real thing for both diagnosis and evaluation. The rationale for this approach was that most professors have, over time, established large item pools from which they draw, in some more or less random manner, to make up any quiz. In addition to their own item pools, many instructors use items suggested from the instructor's manual or handbook that accompanies the text being used that particular semester. Additional verification of this approach to test-design was accomplished by looking at the exam files in the fraternity and sorority houses at our campus.

Because of all of the above considerations, we decided to develop an automatic Testing And Scoring System (TASSY).

We felt that TASSY should have the following specifications:

(1) it should be easy to use by both the student and the instructor;

(2) it should allow for immediate feedback to the student;

(3) it should allow the instructor, on demand, to review the progress of a student;

(4) it should allow the student to individualize his request for proficiency;

(5) it should have a high degree of security;

(6) it should meet at least the minimum needs of the registrar for recording grades;

(7) it should allow the student the option of taking an exam for diagnostic purposes or for grade purposes;

(8) it should maintain a record of each student's individual performance for instructor analysis of items.

TASSY'S PROGRAMMING STRUCTURE

TASSY takes the form of a main driving program with several small sub-routines. This structure was necessary because of the constraints of the Conversational Programming System (CPS) with West Virginia University's IBM 360-75. There is a limitation of four pages (each of four thousand bytes) placed on any CPS program. However, through the use of external procedures, a much greater effective program size can be attained (provided that not more than six thousand bytes are in the work space at any time).

The driving program consists of the "welcome" and "exit" lines as well as the calls for all the sub-routines.

When a student enters TASSY, he is first asked if he would like to see some operating instructions for the system. If he replies yes, an instruction sub-routine "HELP" is called. Next, a password routine, "PASS", is called. Here, it is determined whether he is permitted entry to the system. If the password is recognized as that of an instructor, the user has an option of seeing special operating instructions from "MORE HELP" (restricted to only instructor mode). The instructor then has the option of "UPDATE" or "DUMP" (described later).

If the password is recognized as that of a student, a call to sub-routine "GENER" is issued (also described later).

A third alternative is that the password is recognized as a master password, allowing access to control mode. From this mode, a system manager has the option of "UPDATE", "DUMP", "GENER", or "WHO". The system manager has access to any course.

When a user (student, instructor, or system manager) is finished, control is returned to the driving program where a "good-bye" line is printed. Then the system is ready for the next user.

ROUTINE "PASS"

The password routine has the main purpose of determining whether a given user is authorized to be in the system, and, if so, in what mode. A student password is given by the proctor to a student when he enters the testing center. This password varies systematically each hour of the day, and can be reset by the system manager each day or week as necessary.

An instructor's password consists of any combination of up to six letters, numbers, or special characters and is chosen by the instructor. If an instructor's password is recognized, a further check is made on the name entered by the user. After passing both checks, control is returned to the driving program, passing back a code to indicate that this user is authorized for instructor's mode. Also, the number of the course in which this instructor belongs is returned.

If the master password is found, a code is returned to the driving program to indicate that this user is the system manager and is authorized for anything.

All passwords, including the instructor's and the master password can be altered at any time by the system manager.

An added feature of the password check is the activity file (WHO). A record is written to this field when an instructor's password is found or when a user is not permitted entry in any mode, i.e., when he has entered an invalid password. This record contains the password used, the first and last name and ID number as entered by the user, and the date and time of his entry into the password procedure.

It was decided not to record valid student entries for two reasons. First, the number of entries would be great, and second, they are recorded as a part of "GENER".

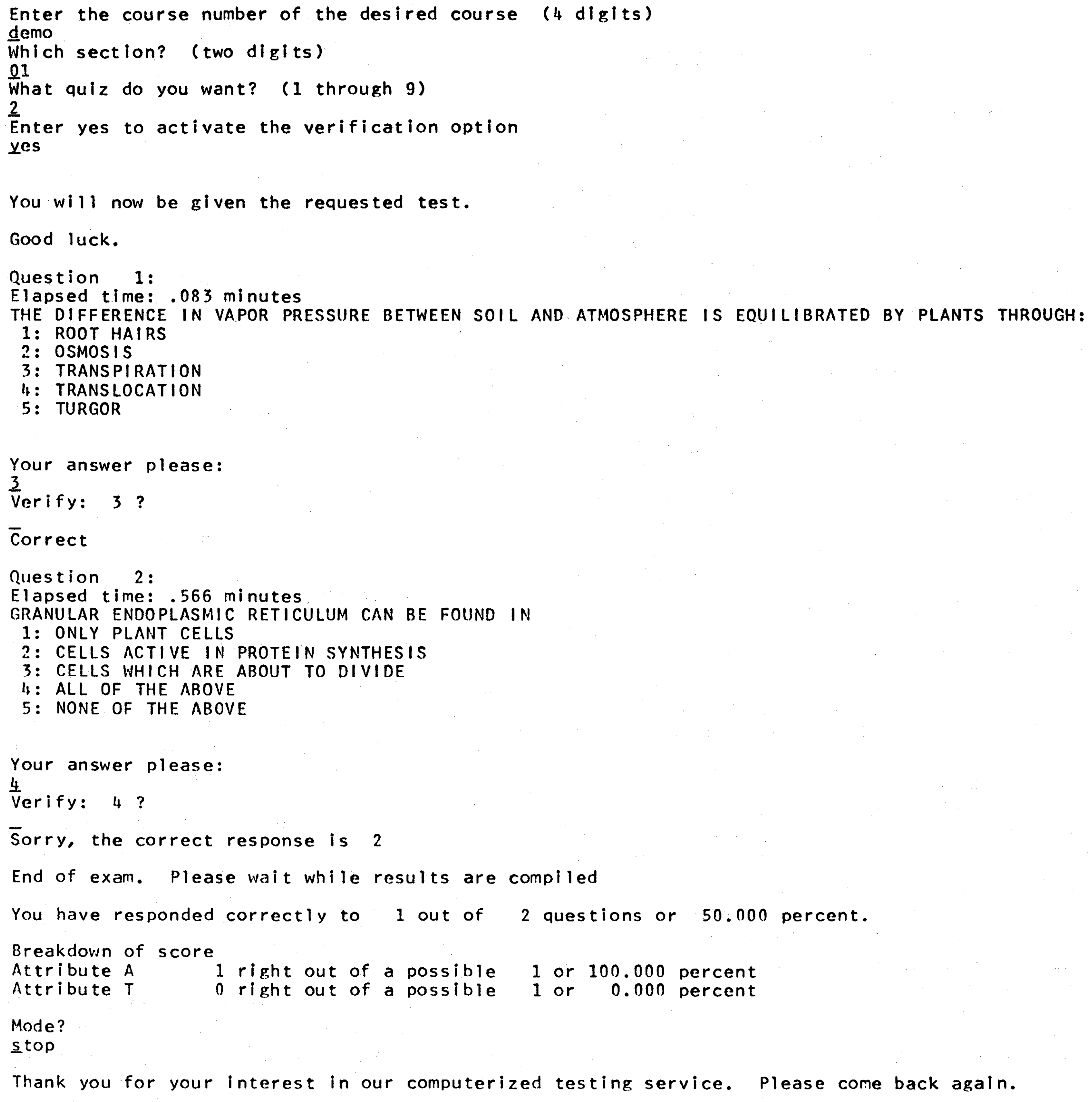
ROUTINE "GENER"

This routine is called to generate, print, score, and record a student's examination. It is probably one of the most important components of TASSY.

Upon entering the routine the student is asked the course, section, and quiz number he desires (his name and ID number have been passed from the driving program).

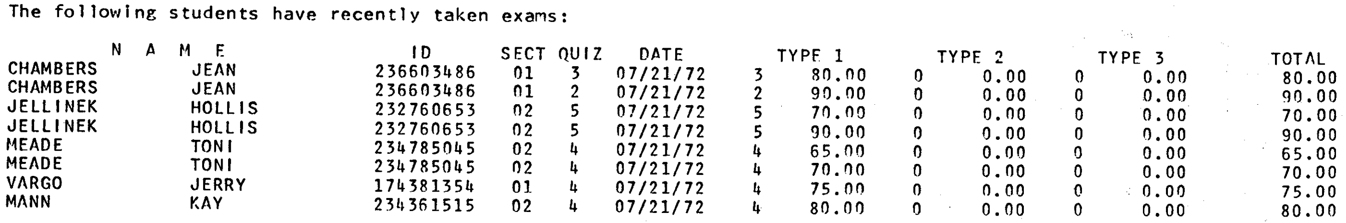
From the course number, the appropriate question file is opened. Then, from a control record in the file, and from the entered quiz number, the type and number of the items to be given is determined. These items are generated from the file at random (except that there has to be the prescribed number of each type designated by the control record).

One rather interesting problem developed in choosing the algorithm for random generating of questions. The records are stored in the file in the order of entry. Along with each record is the attribute of the given question. From the control record, the desired attributes for a given quiz, and the number of each which should be generated, is obtained. In an early version of the system, a random number between one and the number of questions in the file was generated by the built in function in CPS. The corresponding record was then read, and a check was made to determine if the record was of the correct attribute. If not, the number of this record was saved in a vector of 'used' items. If the item had the correct attribute it was printed as described below, and its number was also saved. As each new item number was generated, it was compared against the growing list of used items. This procedure prevents duplication of items on a given test. As the test proceeded from the attribute being used to the next as defined in the control record, the vector of used items was cleared for use by the next attribute.



```
Enter the course number of the desired course  (4 digits)
demo
Which section?  (two digits)
01
What quiz do you want?  (1 through 9)
2
Enter yes to activate the verification option
yes


You will now be given the requested test.

Good luck.

Question   1:
Elapsed time: .083 minutes
THE DIFFERENCE IN VAPOR PRESSURE BETWEEN SOIL AND ATMOSPHERE IS EQUILIBRATED BY PLANTS THROUGH:
 1: ROOT HAIRS
 2: OSMOSIS
 3: TRANSPIRATION
 4: TRANSLOCATION
 5: TURGOR


Your answer please:
3
Verify:  3 ?

Correct

Question   2:
Elapsed time: .566 minutes
GRANULAR ENDOPLASMIC RETICULUM CAN BE FOUND IN
 1: ONLY PLANT CELLS
 2: CELLS ACTIVE IN PROTEIN SYNTHESIS
 3: CELLS WHICH ARE ABOUT TO DIVIDE
 4: ALL OF THE ABOVE
 5: NONE OF THE ABOVE


Your answer please:
4
Verify:  4 ?

Sorry, the correct response is  2

End of exam.  Please wait while results are compiled

You have responded correctly to   1 out of   2 questions or  50.000 percent.

Breakdown of score
Attribute A       1 right out of a possible   1 or 100.000 percent
Attribute T       0 right out of a possible   1 or   0.000 percent

Mode?
stop

Thank you for your interest in our computerized testing service.  Please come back again.
```

Figure 1



```
The following students have recently taken exams:

              N   A   M   E               ID      SECT QUIZ   DATE           TYPE 1          TYPE 2          TYPE 3            TOTAL
CHAMBERS           JEAN                236603486   01    3   07/21/72   3    80.00    0      0.00    0      0.00           80.00
CHAMBERS           JEAN                236603486   01    2   07/21/72   2    90.00    0      0.00    0      0.00           90.00
JELLINEK           HOLLIS              232760653   02    5   07/21/72   5    70.00    0      0.00    0      0.00           70.00
JELLINEK           HOLLIS              232760653   02    5   07/21/72   5    90.00    0      0.00    0      0.00           90.00
MEADE              TONI                234785045   02    4   07/21/72   4    65.00    0      0.00    0      0.00           65.00
MEADE              TONI                234785045   02    4   07/21/72   4    70.00    0      0.00    0      0.00           70.00
VARGO              JERRY               174381354   01    4   07/21/72   4    75.00    0      0.00    0      0.00           75.00
MANN               KAY                 234361515   02    4   07/21/72   4    80.00    0      0.00    0      0.00           80.00
```

Figure 2

As is apparent, if, during generation of items for a given attribute, an item of some other attribute were selected, it would be ignored and its record number would be stored so that this unusable record would not be selected again. It was thought that the machine time used in searching the "used" vector would be less than the I/O time required to keep selecting an unusable record (the records, in this case, must be read before usability can be determined).

What was not realized was that the item vector would grow as fast as it did. As an experiment, the algorithm was changed in a way that the records are read, the test of attribute is made, then the vector is scanned to see if this item has already been used in this test. In this way, many more records are read from the file, but, much fewer comparisons are made in the "used" vector.

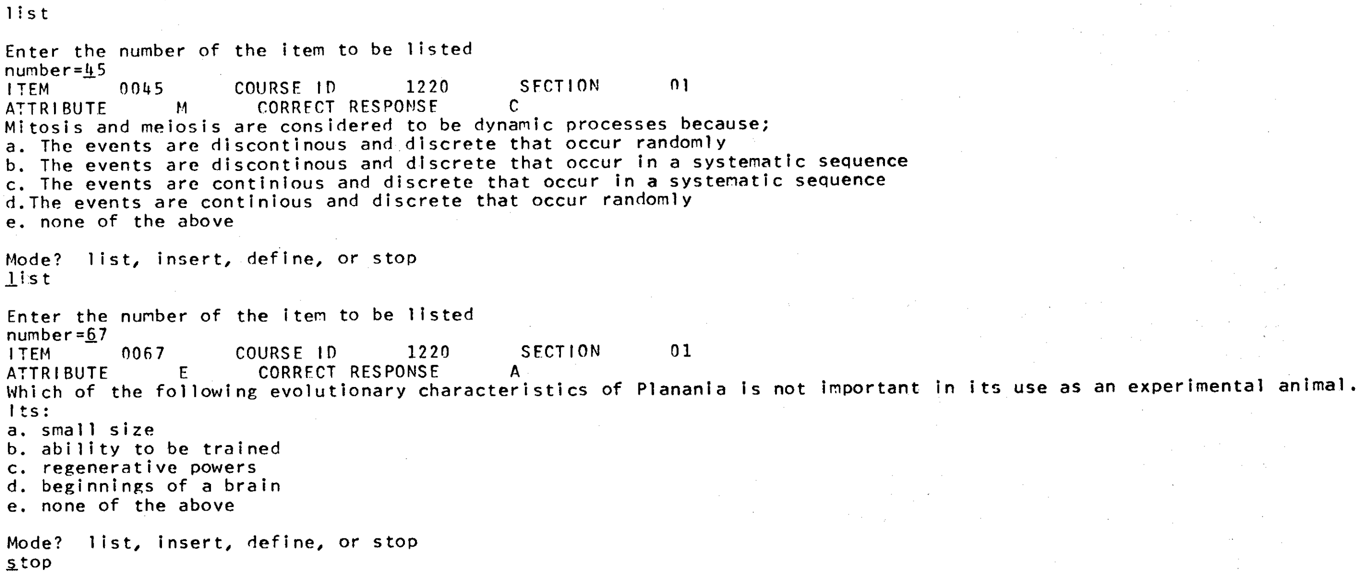
The later of the two methods proved much more satisfactory. In the earlier method, when more than fifteen items of the same attribute were generated, real time between items ran 45 seconds, or more, while the time separation between items in the first part of the quiz (items one through eight) was minimal (on the order of two to six seconds).

In the latter method, the separation time was much more uniform from the beginning of a quiz through twenty items or more and was on the order of two to eight seconds. Needless to say, the latter method has been used since the day that the time differential was noted.

(It is thought that locating various attributes in different physical locations in the file may be a useful way of decreasing item generation time even more than what has been attained by the above change.)

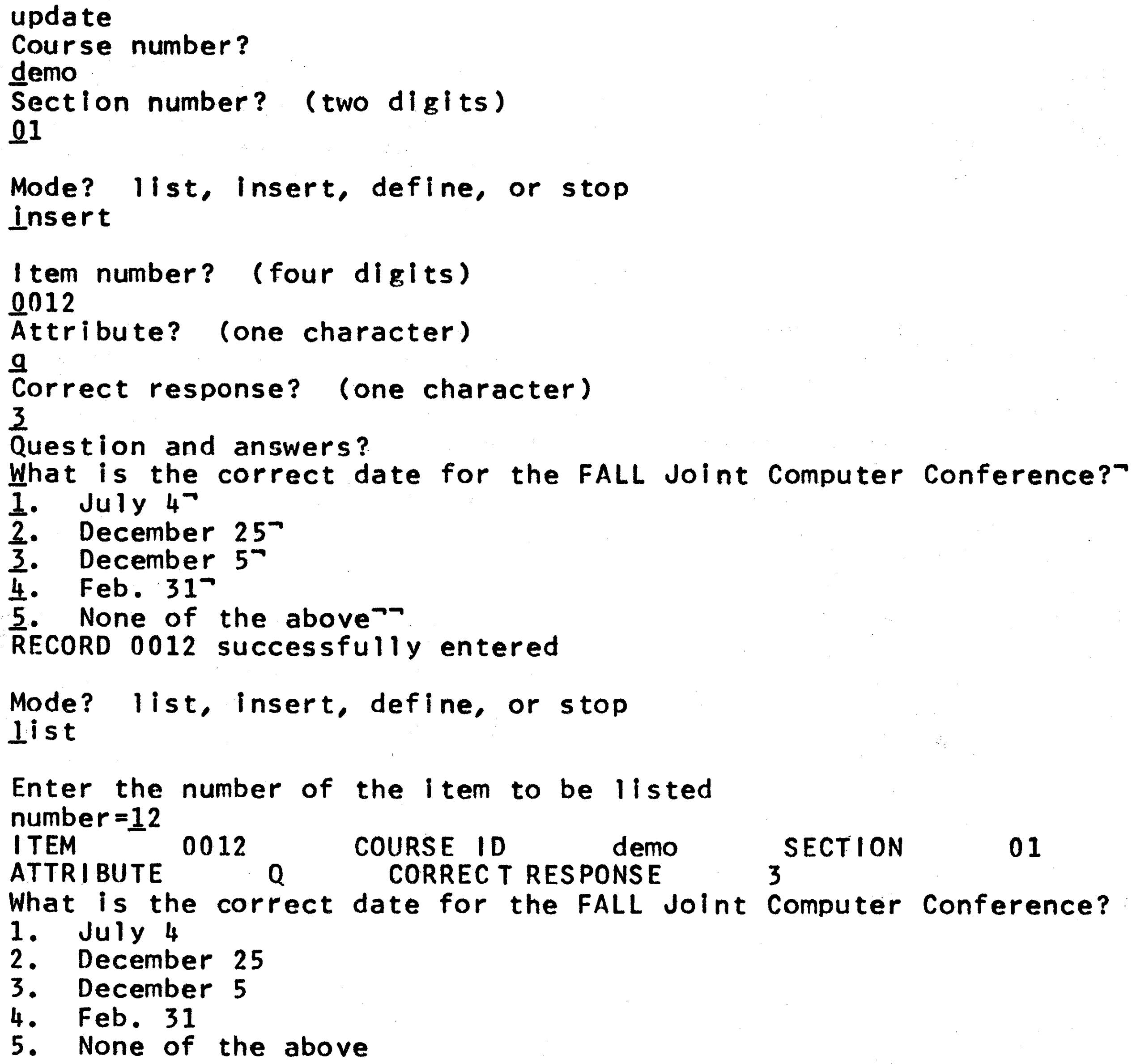
As the items are generated, they are printed on the terminal one by one and a reply is requested. Upon entry of this reply, the student is told immediately whether he is correct. If the response is incorrect, the correct answer is given.

As the test is being given, a record is kept of each question, and, by question type, of the number correct



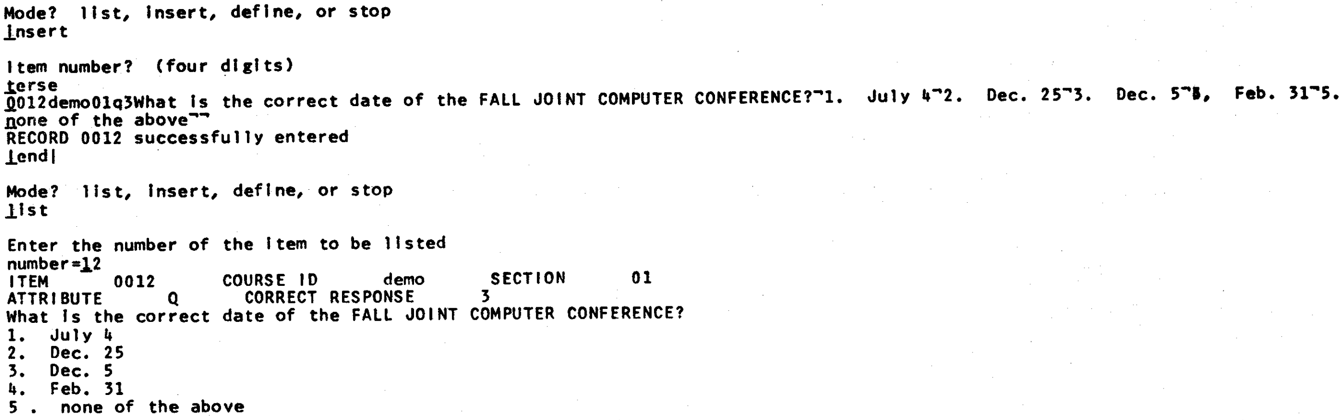
```
list

Enter the number of the item to be listed
number=45
ITEM       0045       COURSE ID       1220       SECTION       01
ATTRIBUTE       M       CORRECT RESPONSE       C
Mitosis and meiosis are considered to be dynamic processes because;
a. The events are discontinous and discrete that occur randomly
b. The events are discontinous and discrete that occur in a systematic sequence
c. The events are continious and discrete that occur in a systematic sequence
d.The events are continious and discrete that occur randomly
e. none of the above

Mode?  list, insert, define, or stop
list

Enter the number of the item to be listed
number=67
ITEM       0067       COURSE ID       1220       SECTION       01
ATTRIBUTE       E       CORRECT RESPONSE       A
Which of the following evolutionary characteristics of Planania is not important in its use as an experimental animal.
Its:
a. small size
b. ability to be trained
c. regenerative powers
d. beginnings of a brain
e. none of the above

Mode?  list, insert, define, or stop
stop
```

Figure 3



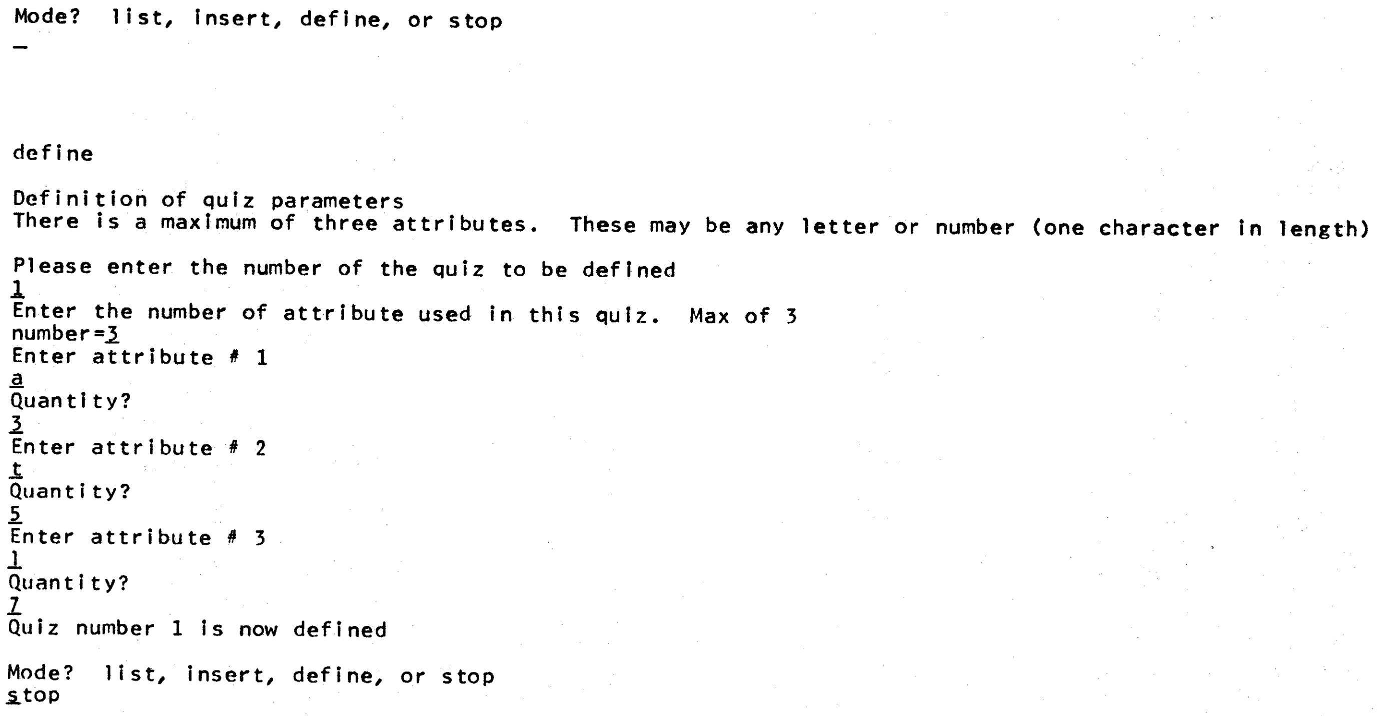
```
update
Course number?
demo
Section number?  (two digits)
01

Mode?  list, insert, define, or stop
insert

Item number?  (four digits)
0012
Attribute?  (one character)
q
Correct response?  (one character)
3
Question and answers?
What is the correct date for the FALL Joint Computer Conference?¬
1.  July 4¬
2.  December 25¬
3.  December 5¬
4.  Feb. 31¬
5.  None of the above¬¬
RECORD 0012 successfully entered

Mode?  list, insert, define, or stop
list

Enter the number of the item to be listed
number=12
ITEM        0012        COURSE ID       demo        SECTION        01
ATTRIBUTE        Q        CORRECT RESPONSE        3
What is the correct date for the FALL Joint Computer Conference?
1.  July 4
2.  December 25
3.  December 5
4.  Feb. 31
5.  None of the above
```

Figure 4



```
Mode?  list, insert, define, or stop
insert

Item number?  (four digits)
terse
0012demo01q3What is the correct date of the FALL JOINT COMPUTER CONFERENCE?¬1.  July 4¬2.  Dec. 25¬3.  Dec. 5¬4,  Feb. 31¬5.
none of the above¬¬
RECORD 0012 successfully entered
lend|

Mode?  list, insert, define, or stop
list

Enter the number of the item to be listed
number=12
ITEM      0012       COURSE ID       demo       SECTION       01
ATTRIBUTE       Q       CORRECT RESPONSE       3
What is the correct date of the FALL JOINT COMPUTER CONFERENCE?
1.  July 4
2.  Dec. 25
3.  Dec. 5
4.  Feb. 31
5 .  none of the above
```

Figure 5



```
Mode?  list, insert, define, or stop
_



define

Definition of quiz parameters
There is a maximum of three attributes.  These may be any letter or number (one character in length)

Please enter the number of the quiz to be defined
1
Enter the number of attribute used in this quiz.  Max of 3
number=3
Enter attribute # 1
a
Quantity?
3
Enter attribute # 2
t
Quantity?
5
Enter attribute # 3
1
Quantity?
7
Quiz number 1 is now defined

Mode?  list, insert, define, or stop
stop
```

Figure 6

and number attempted; i.e., the number of questions of that type which were on the quiz. At the end of the quiz, the student is given the totals of questions correct and attempted and the breakdown of this information by question or attribute type. (See Figure 1)

Before control is returned to the driving program, a student record is written onto a file, indicating the name, ID number, section number, quiz number, date, time, and subscores, and total score on the examination. (See Figure 2)

ROUTINE "DUMP"

The "DUMP" routine is used by the instructor to print the records in the student file. In one sense "DUMP" keeps the instructor's grade book. The formatted file gives the instructor the student's name, his student number, the date the quiz was taken, the number of the quiz, section number, and a percentage correct breakdown for each attribute and percentage correct for the total quiz.

The LIST function will list a requested item from the file. In the LIST is the correct response, question type (attribute), the question, and the distractors. (See Figure 3)

INSERT is the converse of LIST. It allows the instructor to replace or insert an item in the file.

There are two versions of INSERT. In the more commonly used version, the user is prompted before each entry. After each prompt the user enters the information requested. (See Figure 4)

It was found that this method is time consuming and

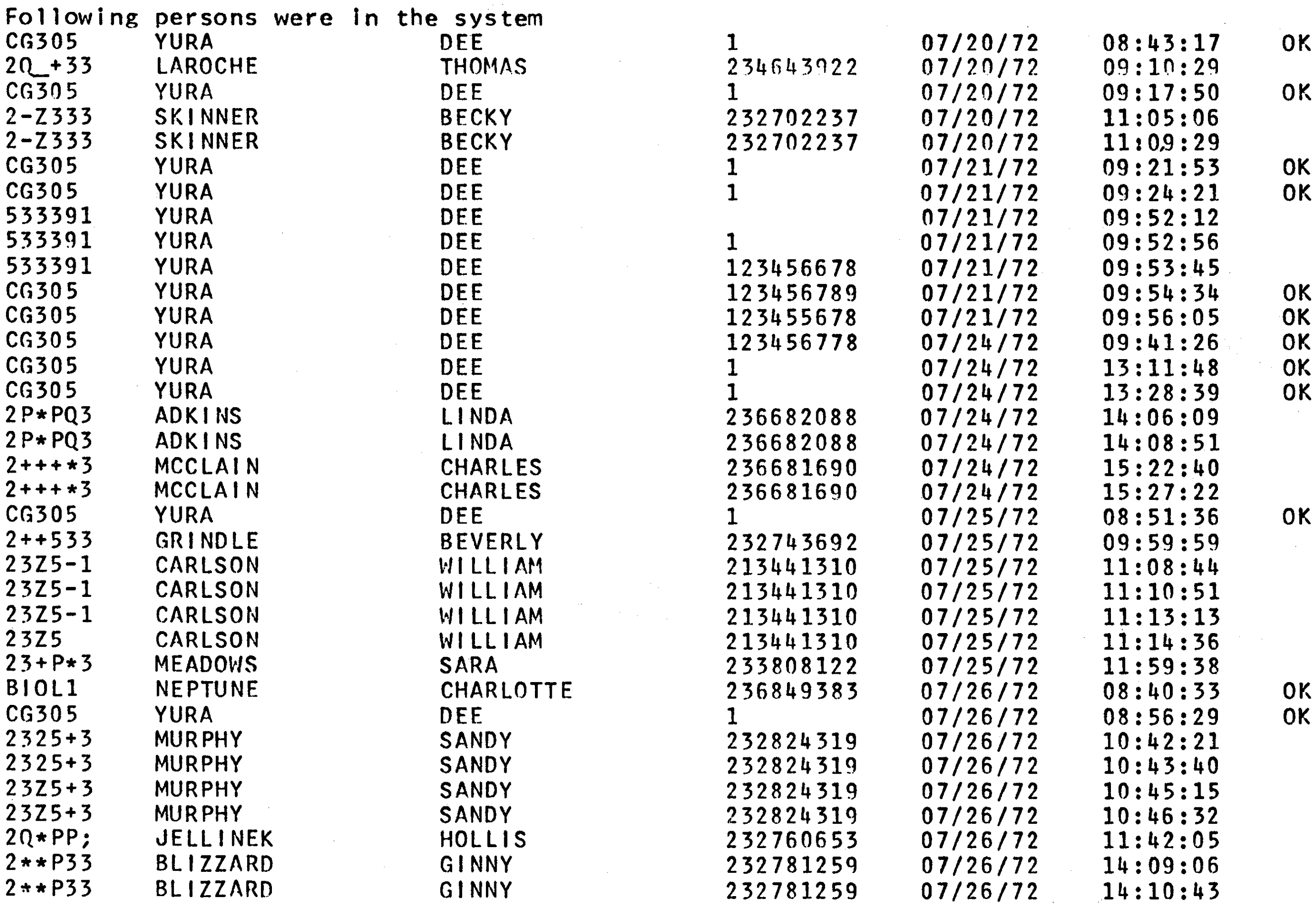
At present, the records are printed in chronological order. However, in later versions of TASSY the instructor will have the ability to have the records sorted for his convenience. (See Figure 2)

ROUTINE "UPDATE"

"UPDATE" is a routine for file maintenance of the question file. The user has the following options: LIST, INSERT, and DEFINE.

somewhat boring for the experienced user, especially when large numbers of questions are being entered. Accordingly, a 'terse' mode of INSERT was developed. When this mode is requested, no prompts are given. It assumes that the user knows the record structure and the entire record is entered at one time. The basic difference between the two methods is that the information (item number, attribute, correct response, course number, and section number) must be provided to the system in the correct order without prompts, when the 'terse' mode is used. (See Figure 5)

In both modes the user must learn only one special character. This is the 'not' (¬) sign which is used as a separator between the questions and its answers, as well as between the answers themselves. It should be noted



```
Following persons were in the system
CG305     YURA                DEE                 1            07/20/72    08:43:17    OK
2Q_+33    LAROCHE             THOMAS              234643922    07/20/72    09:10:29
CG305     YURA                DEE                 1            07/20/72    09:17:50    OK
2-Z333    SKINNER             BECKY               232702237    07/20/72    11:05:06
2-Z333    SKINNER             BECKY               232702237    07/20/72    11:09:29
CG305     YURA                DEE                 1            07/21/72    09:21:53    OK
CG305     YURA                DEE                 1            07/21/72    09:24:21    OK
533391    YURA                DEE                              07/21/72    09:52:12
533391    YURA                DEE                 1            07/21/72    09:52:56
533391    YURA                DEE                 123456678    07/21/72    09:53:45
CG305     YURA                DEE                 123456789    07/21/72    09:54:34    OK
CG305     YURA                DEE                 123455678    07/21/72    09:56:05    OK
CG305     YURA                DEE                 123456778    07/24/72    09:41:26    OK
CG305     YURA                DEE                 1            07/24/72    13:11:48    OK
CG305     YURA                DEE                 1            07/24/72    13:28:39    OK
2P*PQ3    ADKINS              LINDA               236682088    07/24/72    14:06:09
2P*PQ3    ADKINS              LINDA               236682088    07/24/72    14:08:51
2+++*3    MCCLAIN             CHARLES             236681690    07/24/72    15:22:40
2+++*3    MCCLAIN             CHARLES             236681690    07/24/72    15:27:22
CG305     YURA                DEE                 1            07/25/72    08:51:36    OK
2++533    GRINDLE             BEVERLY             232743692    07/25/72    09:59:59
23Z5-1    CARLSON             WILLIAM             213441310    07/25/72    11:08:44
23Z5-1    CARLSON             WILLIAM             213441310    07/25/72    11:10:51
23Z5-1    CARLSON             WILLIAM             213441310    07/25/72    11:13:13
23Z5      CARLSON             WILLIAM             213441310    07/25/72    11:14:36
23+P*3    MEADOWS             SARA                233808122    07/25/72    11:59:38
BIOL1     NEPTUNE             CHARLOTTE           236849383    07/26/72    08:40:33    OK
CG305     YURA                DEE                 1            07/26/72    08:56:29    OK
2325+3    MURPHY              SANDY               232824319    07/26/72    10:42:21
2325+3    MURPHY              SANDY               232824319    07/26/72    10:43:40
23Z5+3    MURPHY              SANDY               232824319    07/26/72    10:45:15
23Z5+3    MURPHY              SANDY               232824319    07/26/72    10:46:32
2Q*PP;    JELLINEK            HOLLIS              232760653    07/26/72    11:42:05
2**P33    BLIZZARD            GINNY               232781259    07/26/72    14:09:06
2**P33    BLIZZARD            GINNY               232781259    07/26/72    14:10:43
```

Figure 7

that this is the *only* place in the entire system where a user has to learn a new symbol. All other commands to the system are in natural language and are very straightforward.

The DEFINE function is used to set up the control record with the quiz definitions. This includes the number of items to be on a given quiz and the breakdown for each type. This record is used by "GENER" when generating the quiz. (See Figure 6)

ROUTINE "WHO"

This routine, available only from control mode, is used to print the system activity file generated by "PASS." It shows who has entered the system and whether or not their password was accepted. The date and time are also available. (See Figure 7)

The value of this routine is to check on activity, especially if an instructor thinks his password is no longer secret.

MESSAGE ROUTINES

Message routines have been implemented. These allow communications from the instructors to the system manager through a file. It is thought that this feature may be valuable for reporting any difficulty to the system manager or for leaving suggestions for improved function.

CONCLUSIONS

Our interest in developing TASSY was to explore the problems and potentials of using the computer in the educational process. TASSY served that purpose in many ways. Our first concern was the problem of software development and record design. We originally designed the question record to be 500 characters long. We found that this is too short. Our next version of TASSY will have the ability to hold a question and associated distractors totalling 1,000 characters on each record.

At the time of this writing we are still not sure what optimal student record should look like. We estimate that the student record should have the ability to record a minimum of 75 items, the sub-scores from 10 attributes, and the total score, in addition to the necessary identification date mentioned in the system description.

A second problem we wanted to evaluate was the feasibility of operating under the auspices of a large central computer using telephone communication. Under the best circumstances, our experience has indicated that we not try it again.

Our experience was not unlike that of anyone else who has had to rely on the telephone system and someone else, i.e., the central system, to do the work for them. A third problem we encountered was that our system has had to rely on the telephone system and some else, i.e., the central system, to do the work for them. A third problem we encountered was that our system became fair game to students who would try to "break in" and look at the answers and the system. The computer center staff developed a special software "lock" for us that was, in effect, a self-destruct button. If *any* tampering was attempted, a system error was generated, duly logged, and the program disappeared. (See Figure 7) At times this security feature was inconvenient, but we felt the trade-off for security well worth it.

We have decided to develop TASSY to operate outside of the University's central computer because of the cost of maintaining enough core and disc space on line. We estimate that the cost of a 16-terminal system would be almost double that of having our own mini-system.

We are also concerned with the reactions of faculty, teaching assistants, and students. The students and teaching assistants liked TASSY very much. The students did not feel that the computer de-personalized them. In fact, most of the students felt that TASSY represented a meaningful step on the part of the faculty to meet their needs. The teaching assistants were overjoyed because the most boring 10 percent of their work assignment was removed. The faculty agreed with the idea, and liked the potential of the TASSY system. When the system became a reality, they wanted to use it as little more than a slow test printer. However, with some handholding and encouragement, and favorable results from the prototype experiment, the faculty have begun to use all of the system capabilities.

Several additional problems arose as a result of our efforts. The faculty had difficulty in developing good test items. Traditional item analysis methodology is only partially useful. Because of TASSY's ability to generate so many different tests, item validity and internal consistency become difficult values to compute. We found that with an item pool of 500 items equally split over 5 attributes, the probability of getting the same test with the same order of items is 1 in $(10!/100!)^5$. Needless to say, sample size for each test is rather small. Our experience with TASSY, plus communication with other researchers, indicates that this problem will be with us for a long time to come.

A second, and even more serious, problem became apparent as we began to develop the diagnostic and prescriptive capabilities of TASSY: namely, once the instructors had detailed data on the student's deficiencies, they didn't know what course of action should be taken except in the most general terms. This has caused some embarrassment.

Our future plans for TASSY include enhancing its response repetoire by adding the ability to recognize single answers and formulas in a manner similar to PLANIT. We also hope to give the instructor his choice of scoring modes besides the traditional, rights, rights-wrongs, etc. Finally, we hope, in the not too distant future, to be able to add some graphics capabilities to the system.

Because of the modularity of TASSY and its relatively simple-minded approach to testing and feedback we feel that it would be implemented readily by individual instructors at almost any level of college instruction. We hope that, as our experience with TASSY grows, we will be able to develop a product which will not only be easy to use, but will also be cost-effective enough so as to warrant serious consideration.

ACKNOWLEDGMENTS

The author would like to give thanks to Mrs. Rita Saltz for her invaluable editorial assistance and to the West Virginia University Computer for their patience.

REFERENCES

1 F B BAKER
Computer-based instructional management systems—A first look
Review of Education Research Feb 1971 Vol 14 No 1 pp 51-70
A C KELLEY
An experiment with TIPS—A computer aided instructional system for undergraduate education
The American Economic Review 1968 No 58 pp 446-457

A comprehensive question retrieval application to serve classroom teachers

by GERALD LIPPEY

IBM Corporation
San Jose, California

INTRODUCTION

CTSS (Classroom Teacher Support System) was developed to aid teachers. The concept consists of retrieving questions according to specified attributes from a centralized data bank, assembling them into tests or exercises, and scoring student answers. Since scoring mark-sense answer sheets is a well-understood and widespread application, the emphasis was placed on solving systems problems related to producing lists of questions which meet the teacher's needs as he perceives them. To achieve this, the system permits items to be classified along several dimensions so that they can be selected by the computer according to criteria set by the teacher requesting a test. (The word "test" is used here to designate a list of questions, regardless of how it is to be used by the teacher who receives it.)

CTSS enables many teachers to share a collection of questions; thus, they all benefit from the advantages of specialization. Such an application has the potential of providing a teacher with access to high-quality questions; freedom from the clerical chores of test construction and scoring; a new test, tailored to his needs, for each occasion; and comparative data based on previous student responses.

Exploration of this concept began in IBM's Advanced Systems Development Division in 1968. In 1969, a joint study agreement was reached between IBM's Systems Development Division and the Los Angeles City Unified School District to develop a prototype application. System functions were specified jointly: IBM developed the computer programs, and the school district prepared an initial collection of 8000 questions in U.S. history and took responsibility for all operational aspects.

Objectives of the joint study were to

1. Confirm decisions related to functional system operation, e.g., communications procedures, forms, reports, retrieval options, item revision procedures, test modification provisions.
2. Identify problems associated with development of item pools, e.g., item classification, cost of preparation, adequacy.
3. Discover how classroom teachers would use questions when they were conveniently available, e.g., testing, drill, discussion.
4. Gain quantitative information on usage, e.g., frequency of use, length of tests, requirement for data bank size.

During the first half of 1969, functional specifications were established. Programs were coded during the second half, while the first item collection was being prepared. Systems testing began in January 1970, with six teachers in one school. CTSS slowly phased into operational use as teachers at various schools have been gradually added during the last couple of years. There are now over 200 history teachers using the system in Los Angeles schools, and several other educational institutions have installed it.

TEACHER SERVICE

CTSS is intended to be entirely under the teacher's control. It may be used or not as the teacher sees fit. The system is free of any particular philosophy of testing or other use; it is the teacher's prerogative to use it in any fashion that satisfies his needs. Questions have been used for quizzes, homework assignments, final exams, drill, review, classroom discussion, and material for special student projects.

Although it can be used with essay and short-answer questions as well, CTSS was intended for objectively scorable (multiple-choice, true-false, and matching) questions. This decision was made to encourage machine-scoring in order to collect data to help identify unsatisfactory items. Within this framework, some features were included to accommodate item format variations: Items consisting of several questions preceded by a paragraph or table are acceptable. Also, special print control provisions permit item authors to specify overprinting of text lines (e.g., for underlining words) and to control the splitting of long items between test pages.

Item collections are maintained on disk storage. Teachers submit requests for questions on optically scanned test-request forms, which are sent directly to the computer center through the district's internal mail. This input is batched, run each night, and the resulting tests placed in the school mail the next day. Scoring is handled in the same fashion.

Item specification

During the design of CTSS, primary attention was given to item selection. So teachers can conveniently construct tests which meet their needs, the system permits questions to be classified in several ways. Although specific questions can be requested, teachers usually request questions by specifying attributes associated with them.

Questions in a broad subject matter area (an "item collection") are classified at the least into major subject matter "categories" and at most along four additional dimensions. The category classification may be based on behavioral objectives or not, depending upon the item collection designer's wishes. It may also be structured in hierarchical levels. During the retrieval process, items are selected from both those in the category specified and those in all categories subordinate to the one specified. There may be up to five hierarchical levels in the category classification defined.

Other classification dimensions can include an assigned difficulty level, behavior level (knowledge vs. application of knowledge), keywords, and several special flags. Some dimensions (e.g., keywords) permit the item classifier to assign more than one value to each item. Dimensions which can have a large number of values (e.g., category) are coded numerically, so that, with the aid of an index, they can easily be specified on an optically scanned test-request form.

Specifications for questions are entered in "request blocks" on the test-request form. Each block consists of several fields in which the teacher specifies the attributes and the quantity of a group of items desired. While items are normally selected by attribute, a request block may be used to specify the unique identification number of an individual item desired. Thus, a test may be constructed which contains a specified number of questions in each of several categories with the desired mix of other characteristics, as well as some specific items which the teacher knows from experience and wants to include.

Tests

The result of processing a test request is a list of questions identified by the teacher's employee number and a two-digit test number assigned for that teacher by the system. The test thus produced is stored by CTSS and labeled "generation 1."

The teacher may then modify the test by requesting the system to add or delete items. To accomplish this, a test-request form is filled out which references the test number and specifies the items to be added in the same way that an initial test is requested. Items to be deleted are indicated on another field in the form. A new list of questions with the same test number will be created and labeled "generation 2." This process can be repeated until the list of questions satisfies the teacher. Only the most recent generation of a test is remembered by the system. A teacher may have up to twenty such tests retained simultaneously.

The teacher may specify on the test-request form that the test be printed on a reproduction master. He may also request up to nine different versions of the test with the items appearing in a different sequence on each.

Each time a test is printed, two associated reports are produced. An Item Characteristics report provides the answer key and informs the teacher how each item has been classified for retrievel. It may also provide references to two resources which contain information related to the content of each item. The second report repeats the teacher's request and indicates the items retrieved in response to each request block.

CTSS will score student answer sheets when the teacher so desires. Since the test has been remembered by the system, it is not necessary for a teacher to submit a scoring key. Several scoring options are available for identifying students, suppressing or adding questions, and partitioning reports. The usual scoring reports are sent to teachers. Scoring procedures and reports will not be discussed.

The system was designed so that on-going, everyday service could be provided without the need for judgment by anyone other than the teacher concerned. Probably the most important consequence of this objective was the attempt to anticipate input errors of various kinds and, whenever possible, respond automatically by addressing an explanatory message directly to the teacher.

SUPPORT ACTIVITIES

The teacher service described above is supported in several ways. Direct daily support is provided by the internal mail service and the data processing center. At the center, request forms are batched and read by an optical mark reader; processing is accomplished at night; and output to each teacher is manually matched to its request form prior to its return to the teacher.

CTSS is, however, not adminstered by the data processing group. Rather, the data processing center performs a service function, while operations are monitored and managed by education-oriented personnel referred to as CTSS "coordinators." This arrangement dictates that the points of contact between the data processing center and others be well defined, so that the computer center can regard all CTSS jobs as routine production. Consequently, input from, and output to, both teachers and coordinators is handled according to well-established procedures.

Service support

Coordinators have two areas of responsibility—one related to operational teacher services and the other related to item pools. In the teacher service area, new users of the system may obtain coordinator assistance getting started and, subsequently, in understanding errors that they make. Coordinators also supply teachers with the optically scanned request forms.

Since a user identifies himself to CTSS on the test and scoring request forms by employee number only, a file of teacher names and locations, the "teacher file," is required to automatically address the title page of tests and scoring reports. A teacher must be registered on this file prior to using CTSS. One chief responsibility of the coordinators is maintenance of the teacher file.

The file in which tests are stored, called the "active list," can also be influenced by coordinators. When a new test is generated, it is automatically added to the active list; when a test is scored, it is deleted. Since many tests are never scored by CTSS, the active list would continue to grow indefinitely unless old tests were removed. Old tests are identified by assigning an "activity date" to each test when it is created. This is reset to the current date whenever a new generation is produced or a scoring request is not successfully processed due to input error. The activity date is used to purge old unscored tests from the file. A "time-out cancellation" program removes from the active list tests whose activity dates precede a cancellation date set by a coordinator. When a test is timed-out, a notice is sent to the teacher concerned, informing him that it is no longer available for modification or for scoring against the answer key retained in the system. Time-out cancellation is initiated periodically by coordinators.

As tests are produced and scored, statistics on system activity are accumulated. A "system statistics file" contains counters which cumulate data for two dozen kinds of user activity. For example, the number of test requests rejected due to input errors, the average number of generations produced, and the average number of questions requested on tests are accumulated. The system statistics file contains this information for two durations (a long and a short time period); the data is maintained for each item collection; and it is classified according to which of 13 teacher groups the corresponding users belong. Coordinators may reset the system statistics file counters when they wish the accumulation process to begin again.

In addition to direct contact with teachers, coordinators monitor system usage by looking over summaries produced by each test production and scoring run. There is also a report generated by the time-out cancellation program, which summarizes the number of tests in the active list, tests having scrambled versions, and tests removed from the active list by the time-out routine. Longer term activity is observed by drawing activity reports from the system statistics file described above.

Item pool support

The second area of coordinator responsibilities involves the item collections themselves. To use an item pool, teachers must understand how it has been classified, and they must have access to at least the index which defines subject matter category numbers and perhaps to other indexes which have been constructed. Coordinators are responsible for communicating this type of information to teachers.

Item collection maintenance is another important coordinator job. Typically, large collections of questions are made available before they have been thoroughly edited and field tested (otherwise, the development investment would be too large). Thus, one begins with relatively poor-quality items and depends upon a long range revision process during usage to improve the questions. As teacher comments and scoring data become available, items are repaired. Teacher reactions to poor-quality items, while negative, have not turned out to be a serious problem. Indeed, teachers sometimes appear to experience satisfaction when they discover and report items needing correction.

The best source of item revision information appears to be the teacher. A second source has been provided in CTSS by cumulating item usage data in an "item statistics file," associated with each item collection. The item statistics file retains information on the number of times each question appeared in a test, or was deleted from a test or suppressed from scoring by a teacher. The file also contains student-supplied data obtained when questions are machine-scored, such as the number of responses to each option and a central tendency for discrimination index.

A coordinator can obtain an "item statistics report" based on information in the item statistics file. The philosophy behind this report is that by appropriate selection of item statistics thresholds he can obtain a list of those items most likely to need revision. A coordinator may set thresholds on high teacher rejection rate, low average discrimination, unusually heavy use of a distractor, and very high or low measured difficulty level. He may also require that some minimum number of tests has been drawn, or answer sheets scored, to cause an item to be eligible for these tests.

Finally, it is the coordinator's responsibility to oversee the creation and supervise the installation of new item collections. When a new item pool is to be constructed, many decisions need to be made: a character set, i.e., those characters which are to be allowed in item text, must be chosen; the kinds of items which are permissible must be determined; the dimensions of item classification must be designed; and the method of transmitting all the necessary classification information to teachers must be planned.

There appears to be significant value in having large item pools. Teachers do not usually wish to encounter the same few items over and over. A large collection is especially useful when multiple tests are requested to cover the same material. It also enables the collection designer to include several approaches to subject matter; this is essential if the collection is to be shared by users having a variety of pedagogical styles. Finally, it appears to be helpful if teachers regard an item collection as essentially infinite in size and constantly changing, and do not, therefore, have a desire to deal manually with the entire collection at once. Experience from CTSS indicates that about 30 questions per class hour should be regarded as a minimum item pool size; 50, as more desirable; and, perhaps, about 70 as the number beyond which the cost begins to exceed the value.

SYSTEM DESIGN

CTSS was designed as a prototype because it addresses a new application area. Many of the design decisions were influenced by this. Perhaps the most obvious effect on system design was the effort to include features whose utility was questionable in order to establish their value through experience.

On the other hand, an effort was made to design low cost into the system framework. Thus, on-line terminals were rejected in favor of internal mail service, and keypunching of teacher requests is avoided by using optically scanned input forms. Costly human intervention is further reduced by sending error messages directly to teachers. Also, teachers are discouraged from requesting reproduction masters or scrambled versions unnecessarily by preventing further modification of a test once either of these more expensive printouts has been produced.

Program design priorities (from high to low) were as follows: (1) ease of coding and testing, (2) ease of modification and maintenance, (3) low storage requirements, (4) low execution time. Prototype design specifications included handling several item collections with about 10,000 items in each. The system was programmed in PL/I to run under the IBM/360 Operating System in a 74K byte partition.

A highly modular programming approach was chosen. Communication between programs is accomplished through files which are either permanently established or used as temporary interfaces. Dividing functions into a series of separately executable programs simplified programming and testing. More importantly, this made it easier to modify the system, particularly when additional features were later inserted. The permanently established files will be outlined next, followed by a brief summary of the major runs available.

Files

CTSS includes two types of permanent files: those which are item collection-independent and those which are item collection-dependent. An "item collection-independent" file is required by the system only once, irrespective of how many item collections are supported. An "item collection-dependent" file is required to be present for each item collection. The main permanent files and their contents are itemized in Tables I and II.

Access to items

Item manipulation is, of course, the key element of a test construction system. Ease and efficiency of item retrieval and item revision depend upon the item file organization. In CTSS, all of the selection decisions concerning which items are to appear on a test are made by consulting the classification file. This file contains item attributes, but no item text, and is therefore very small and easily referenced compared to the item file. During the run that produces tests, the item file is referenced only when it is necessary to format the test

TABLE I—Principal Item Collection-Independent Files

File Name	*Contents*
Course File	Record for each item collection: Identification, location, and parameters
Teacher File	Record for each teacher: identification and address; identification of each active test, its activity date, and its location in the active list
Active List	Record for each active test: identification of items, answer key, and status information; identification of last 50 items deleted during previous modifications; location of version file record, if any
Version File	Record for each active test having scrambled versions: scrambling keys
System Statistics File	Record for each item collection: two sets of system usage information for each teacher group

for printing. (Similarly, the active list contains records of specific tests by recording item identifications instead of the item text itself.)

It was decided early in the design phase that, while several dimensions of item specification would be available to teachers, one important dimension would be emphasized and retrieval optimized around it. As a result, items are ordered by subject matter category number in the item collection-dependent files, making hierarchical selection in this dimension easy to implement. Instead of building and maintaining inverted files for other dimensions, the classification file is simply scanned within the category designated for items which meet other specified criteria. To discourage teachers from too often requesting a scan of all items in a collection for those items which have some other attribute, they are prevented from initiating such a search with a single request block. By requiring that teachers enter a category number in each request block used and limiting the range of a search initiated by one request block to the highest level of the hierarchical category classification system, CTSS can require that several request blocks be used to cause a search over the entire item collection. Consequently, though it is possible to initiate a scan of the whole classification file, a teacher must go to some trouble. This compromise between meeting user needs and discouraging use of unnecessary computer time has so far proven satisfactory.

TABLE II—Principal Item Collection-Dependent Files

File Name	*Contents*
Classification File	Record for each item: item attributes and location in the item file
Item File	Item text
Item Statistics File	Record for each question: item usage data

The item file itself consists of 80-character card images, where each image contains one text line of a printed item and a unique identification number. No attempt was made to compress text by coding blanks. This file serves as the master file of items; there is no duplicate file of punched cards. When cards are desired to aid in changing item text, they are punched.

The item file is normally accessed differently for file maintenance than for test construction. Items are retrieved from the item file for tests by direct access, but the basic item file maintenance run is a sort-merge procedure which rebuilds all of the item collection-dependent files. Item additions, deletions, and substantial modifications may be accumulated and this run executed infrequently. Between such runs, it is possible to prevent specific items from appearing on tests and to change specific item cards in the file by direct access.

TABLE III—Principal Runs for Coordinators

Run Name	*Function*
Time-Out Cancellation	Remove old tests from the active list
Print Teacher File	List the teacher file and active list
Teacher File Maintenance	Add, delete, or modify teacher identification data
Print Activity Report	Produce report from system statistics file and reset counters if desired
Print Item Statistics File	List item statistics
Print Item Statistics Report	Identify items having statistics which exceed specified thresholds
Print Classification File	List item attributes
Print Item File	List item text for specified section of the item collection
Item File Maintenance	Add, delete, and replace items, and reset an item's statistics if desired
Item Stop and Change	Tag item so that it will not be available to teachers, or replace item characteristics or text line

Computer runs

Like those for item file maintenance, most computer runs were designed for use by coordinators. The principal runs available to coordinators are listed in Table III. A few other runs are for the programmer during system maintenance or when adding new item collections. The two primary runs, executed daily, are those which service teachers: One produces tests and the other handles scoring. Test production will be discussed here; scoring will not.

Test production

There are two strategies which might be employed for batch retrieval of items: One is to publish a catalog of all items in the collection, from which the teacher selects those he wants; the other is to have the computer select items from the collection according to attributes specified by the teacher. When a large question data bank is involved, it is impractical for the teacher to deal directly with the questions. Furthermore, if, in addition, new items are continually being added and old ones revised, providing teachers with a relevant catalog of items becomes a problem. It would be hundreds of pages long, and the publishing cost would be compounded by the need for frequent revisions to account for changes. For these reasons, CTSS relies instead upon the computer's ability to retrieve by attribute, while still reserving to the teacher his right of final choice.

However, the approach chosen results in another problem. When selection by attribute is used to locate entries in a large data bank, the difference between the quantity desired and the quantity available must be resolved. This problem is easily handled in a conversational retrieval system, because the user can specify attributes and immediately learn how many items are available. If there were more than he wanted, he could tighten up the specifications and inquire again; if there were less, he could loosen them. In a batch retrieval system, some alternate means need to be employed to insure that the user is neither flooded with eligible items nor receives too few. In CTSS, random selection achieves the former and automatic specification relaxation the latter.

If, for example, a request block specifies five questions having certain characteristics and there are 100 that satisfy the criteria, five are picked at random from the 100. CTSS does this by partitioning the 100 questions, ordered by category number, into five groups of 20 items each. One question is selected at random from each group. The stratified sampling prevents too many items from being occasionally picked from a single category when selection ranges over several subordinate categories. There are, of course, many alternative ways to reduce the number of eligible items, but random selection has proven adequate.

If fewer items are found that match a request block's specifications than are requested, some specifications will be relaxed in an attempt to meet the quantity objective. This is consistent with the observation that teachers prefer to receive items, even though they do not meet all criteria specified. In the prototype, behavior level, if specified, is ignored first; then, any assigned difficulty level specification is disregarded. No other type of teacher specification is relaxed.

When more than one request block is used on a test-request form, each is treated separately for item selection purposes. However, no item is included in a single test more than once. In addition, the identifications of items deleted from a test during modification are stored in the active list. Such items are considered ineligible for subsequent generations of the test unless specifically requested.

The test production run consists of a series of programs, each of which operates on a batch of test requests and runs to completion prior to the next program's execution. This run handles both initial requests and requests for modifications to existing tests. Considering only requests for new tests, the primary functions of each program are summarized in the steps below:

1. Convert the format of optically scanned test-request forms to one more useful during program debugging and maintenance
2. Sort test requests by item collection (so that all item files need not be accessible simultaneously)
3. Edit test requests for teacher input errors and allocate space in the active list and version file
4. Select items to appear on tests by referencing the classification file; store identification of items selected in the active list and scrambling keys in the version file
5. Update the system statistics file
6. Prepare tests for printing by retrieving text from the item file
7. Print summary of this run
8. Print tests on paper (spooled)
9. Print tests which are to appear on reproduction masters (spooled)

Backup and recovery

CTSS has an extensive set of backup and recovery procedures built-in to reduce the impact of human or machine errors related to processing. Because all data needed for runs is retained in files, backup procedures need only be able to restore these files. Prior to executing the test production, scoring, teacher file maintenance, and time-out cancellation runs, the teacher file, active list, and version file are automatically copied. If the run does not go to completion, these files are restored. Because the most recent state of the quasirandom number generator used to select items and scramble tests is stored in the active list header, a test production run can be reproduced from restored files. The system statistics file is copied periodically. Every time item file maintenance is run, a copy of the item classification file, the item file, and the item statistics file is automatically made.

CONCLUSION

Probably the most significant systems learning that occurred during prototype operation has been in the support area. The fact that a good deal of attention was directed toward reducing teachers' chores turned out to increase the coordinators' work substantially. For instance, although teachers are encouraged to offer new items and suggestions for improvements, they are not expected to revise questions. Likewise, teachers are not required to submit their names and locations with requests. Such system-provided services resulted in the presence of additional files and more support work for coordinators. The coordinators' activities have required more computer assistance than had been anticipated during system design. In fact, most of the functions added after CTSS was installed were to aid coordinators—either to diagnose teacher difficulties or to maintain files.

The CTSS prototype is available to other educational institutions; programs, documentation, and some of the existing item collections may be obtained from the Los Angeles City Unified School District. Several other institutions have installed CTSS and more data banks of questions are becoming available. Also, systems of a similar nature have independently emerged at various other locations, chiefly in institutions of higher learning. One can infer from the success of CTSS and from the developing interest elsewhere that the use of computers for banking questions and generating tests and exercises is an embryonic application area which will continue to grow.

Furthermore, test generation is a natural component of more sophisticated computer-assisted instructional approaches. A few of the existing automated test construction activities are, in fact, parts of larger computer-managed instruction systems. These more extensive systems usually include pedagogical decision-making elements, such as diagnosis of learner difficulties and prescription of assignments. Some of them enable students to proceed through large units of instructional material independently of each other. Those who wish to begin with a small, simple system and grow toward a comprehensive system may find test construction a convenient starting point, since it can stand alone under teacher control as well as fit into an integrated computer-managed instruction system at a later time.

Computer processes in repeatable testing

by FRANKLIN PROSSER and JEAN NAKHNIKIAN

Computer Science Department
Bloomington, Indiana

INTRODUCTION

There is emerging an increased interest in computer augmented testing procedures. Among those feasible techniques that have proven of particular value is the method of Computer Generated Repeatable Tests (CGRT). This approach to testing, which allows the *repeatable* administration of tests over a body of material, has been previously described.[1] Interest in the method has been high, and frequent inquiries into the nature of the computing processes involved in CGRT have led us to elaborate here on the computer software aspects of the method. In this paper we describe the structure of the test generation and student response scoring programs, we describe important performance improvements, and we discuss some aspects of the problem of developing "portable" or machine-independent computer programs.

The CGRT process was conceived by Donald D. Jensen, now at the University of Nebraska, as a means of avoiding many of the adverse features of conventional testing in large university classes. The typical exam in such classes consists of true-false or multiple choice questions, is administered at one fixed time only, and is given infrequently and over a large amount of material. Often several days elapse before the student receives any useful information on his performance, and often the total score is the only information given, an item that is of little value in guiding further study. Such procedures are disliked by students, who frequently adopt the "loaf-and-cram" pattern of study, and who are subject to considerable anxiety over their performance on the infrequent and all-important big exam.

The CGRT scheme provides reasonable alternatives to these objections to conventional exams. First, the student may be examined *more frequently*, which encourages the student to keep up with his course work. Second, the student may receive *immediate feedback*: when he turns in his answer sheet, he may receive the correct answers and other study aids. And third, the examinations are made *repeatable*. Large numbers of unique but equivalent individualized tests are prepared using a digital computer. The instructor may readily permit his students to be examined repeatedly over the same unit of material. Students may learn from their errors, and return to be tested again over similar material. In this way, the examination is made a vital part of the learning process.

The CGRT method consists of four basic steps: developing pools of test items, producing the individualized tests, administering the tests, and scoring of student responses to the tests. The entire process is described in general terms in Reference 1. Our purpose here is to discuss in some detail the computer processes involved in repeatable testing.

STRUCTURE OF PROGRAMS

Of the four basic steps in the CGRT procedure, the preparation of tests and the scoring of student responses are facilitated by computer. Test preparation is accomplished with a program GENERATOR, and scoring is done by GRADER. All programs are written in FORTRAN. In the following sections, the essential features of the computer programs are described.

Generator

Figure 1 shows the overall flow of program GENERATOR. The input to GENERATOR consists of the pool of items (questions and answers) to be used in formulating individualized tests, and directives describing the structure of the tests. The program reads the pool of test items, verifies that each item is in an acceptable format, and stores the information as a convenient data structure in the program.

A typical item consists of the body of the question, contained on as many records (cards, usually) as nec-

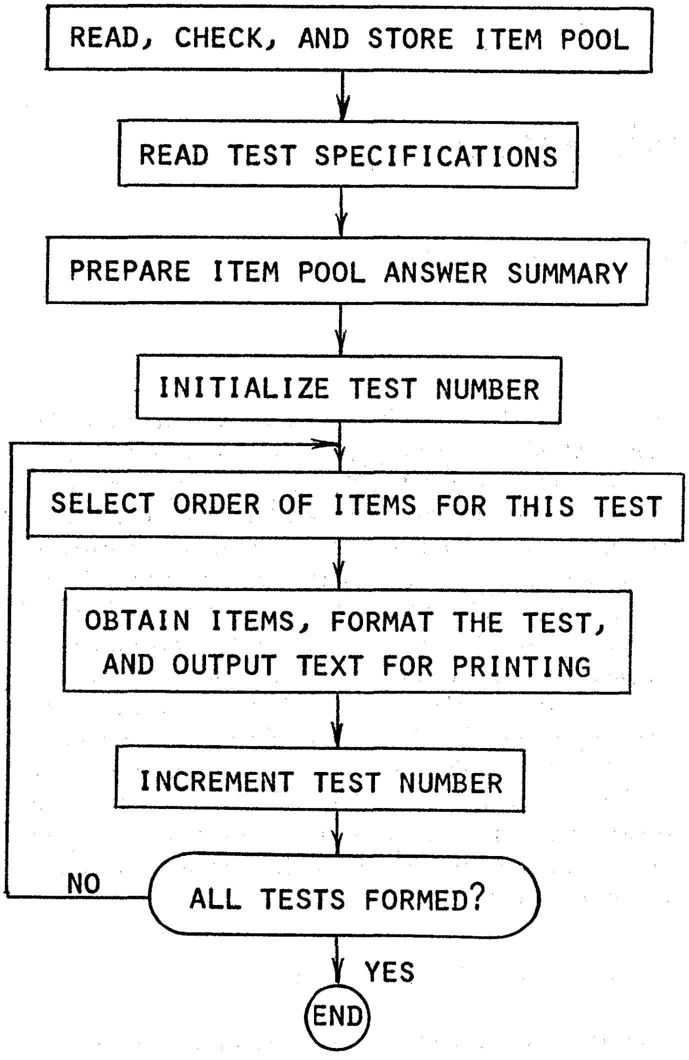

READ, CHECK, AND STORE ITEM POOL
READ TEST SPECIFICATIONS
PREPARE ITEM POOL ANSWER SUMMARY
INITIALIZE TEST NUMBER
SELECT ORDER OF ITEMS FOR THIS TEST
OBTAIN ITEMS, FORMAT THE TEST, AND OUTPUT TEXT FOR PRINTING
INCREMENT TEST NUMBER
ALL TESTS FORMED?
NO
YES
END

Figure 1—GENERATOR program flow

essary, followed by one or more cards of answer information. To facilitate computer grading, the correct answer is placed in a standard position on a card; other supporting answer information (textbook references, etc.) may follow on this or subsequent cards.

The cards are numbered in such a way that the correct card order may be verified, and the answer part distinguished from the question body. Each item is further identified with a numeric code. Typically, to permit the instructor to group his items into sets of items covering similar material, the item identification will consist of a *set number* and an *item number* within the set.

The collection of items to be used in forming a series of individualized tests thus is divided into a variable numbers of sets, each containing a variable number of items. The instructor may assign a specific *weight* to each set. The weight applies to each item in the set, and represents the number of points to be awarded for correctly answering such an item. Further, the instructor may assign a *frequency* value to each set of items, the frequency being a relative or absolute measure of the number of times on a test that the set is to be used for selection of an item. Weights allow control of the point value of items, and frequencies permit control of the number of items used from each set. When weight factors are used, either with or without frequency specifications, some simple rules are imposed to assure a constant number of points on each test.

The pool of items that forms the principal input to GENERATOR may reach a substantial size, perhaps several thousand cards. Some users of repeatable testing find it convenient to maintain their item pools on a master file and manipulate the pools with an updating or editing system.

The item pool is organized by GENERATOR into a list, which is controlled by several tables. An important aspect of the overall efficiency of the CGRT process is that the entire pool is kept in directly addressable memory, since it will be repeatedly accessed in a random fashion. In a subsequent section we discuss ways to operate with the data kept primarily on secondary storage. All items in a set are stored in logically adjacent positions in the list. The list of items is accessed by a vector of pointers to the origin in the list of each item. To distinguish sets of items, another vector contains set pointers to positions in the first vector, each set pointer specifying the index in the vector of the first item pointer for that set. The names of the items and of sets, as supplied by the instructor in his item pool, are not used in the item selection processes; only the position of a set in the pool and the position of an item in its set is relevant. Thus the items may be accessed by position indices rather than by name; this allows a very rapid retrieval of any item.

GENERATOR next reads the specifications for the tests to be produced from the pool of items. Required parameters are an examination unit number, the number of individualized tests, the number of copies of each such test, the number of questions to appear on each test, and a starting value for numbering of the individual tests. Among the optional specifications, in addition to the previously described weight and frequency factors, is the ability to specify randomized or ordered selection of sets during test generation.

Once the test specifications are known, GENERATOR produces a small output file (usually on cards) describing the structure of the item pool, the item identification, the item answers, and certain critical test parameters. This file, typically containing about twenty cards, will permit the later regeneration of the sequence of items (and answers!) on any given test, and thus will permit grading of student responses without the necessity of explicitly preserving the answers for each form. The amount of information which must be maintained between test preparation and scoring is therefore very small.

With these preliminaries attended to, GENERATOR then formulates and prints each test. The selection of items for a test involves two stages: selection of a sequence of sets, and then choosing of an item from each selected set. The instructor will have stipulated either random or ordered set selection, and may have provided frequencies for his sets. Randomized set orderings are effected using a pseudo-random number generator; ordered set selection requires that the sets be used in the order that the instructor presented them in the item pool. Item selection from within a chosen set is always randomized; of course, the item selection is performed without replacement, to assure that no item appears more than once on a given test. The designation of both set and item is achieved conveniently using indices to the set and item vectors previously described.

For later scoring to be performed by regeneration of the answer sequence for each test, it is vital that the pseudo-random numbers used be reproducible. This is achieved by using a simple function of the test number as an initializing value for a random number generator.

Using the indices to the selected items, GENERATOR formats and writes the test onto a file for subsequent printing. See Reference 1 for an example of the format of a printed test. The test consists of a heading section followed by each item, with question body on the left and answer part on the right. The answer material will be removed by the instructor prior to administration of the exam, and will be given to the student upon his completion of the test. As one might imagine, the speed of test generation is very heavily dependent on the speed of the test formatting and output processes. As is shown in a later section, very dramatic improvements in program performance may be obtained by some rather simple (but unfortunately non-standard) manipulations of the FORTRAN output processes.

Grader

Scoring of student responses is performed by GRADER, whose program flow is given in Figure 2. Its first important act is to read and record the information contained on the small answer regeneration decks prepared by GENERATOR. Since the instructor may have prepared tests with several different executions of GENERATOR (and indeed may have changed items between runs), he may supply several answer regeneration decks to GRADER. The principal requirement for the success of this procedure is that no two different tests over an examination unit have the same test numbers. This is easily arranged by the instructor at the time he prepares the tests. For meaningful scoring, the instructor will also see that each test has the same total point score. The answer information is structured in memory in a fashion very similar to that used by GENERATOR for the item pool, except the text of the items is not present during grading. The student typically marks his responses on a mark sense form, which is subsequently reduced to a punch card by an optical form reader. An annoying phenomenon is inherent in many such operations because of inadequacies in form design and reader capability: the mark sense answer form may contain insufficient space for each item to uniquely record the range of possible answer characters, and some doubling up of spaces is required. This imposes a many-to-one

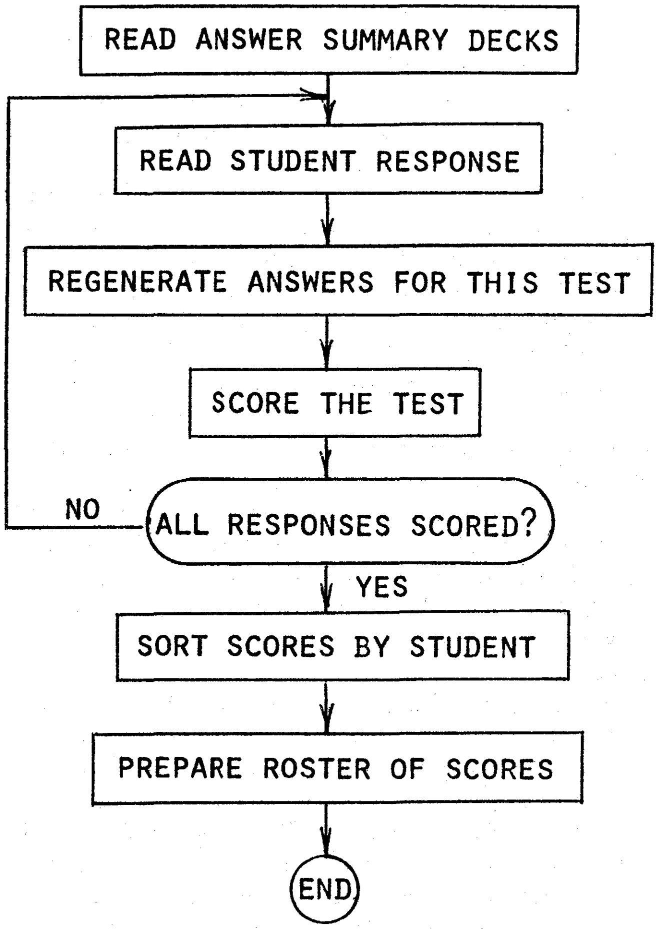


Figure 2—GRADER program flow

mapping on the information that reaches the computer, and requires that the original item pool answers be similarly transformed. This transformation is described to GRADER in a series of FORTRAN data statements, and is imposed on the correct answer information prior to entering the scoring section of the program.

A student's response to a test thus typically reaches the computer as a card containing his name or identification number, the individual test number, and his (possibly transformed) answers to the questions. Using the test number and the information supplied with the appropriate answer deck, GRADER performs an item selection identical to that done by GENERATOR when the test was prepared. With the sequence of correct answers at hand, GRADER scores the student's response, weighting each item appropriately. When all student responses have been scored, GRADER sorts the results according to student identification, and prints a roster of the scores. A student may have taken several tests over the same examination unit; his scores will be listed together on the roster, ranked either according to score or test number, as dictated by the instructor.

Several special features associated with the grading process are available. We mention them only briefly. A number of computer programs are in use for cumulative recording of examination scores and subsequent assignment of a course grade. We have a rather primitive but useful item analysis routine which assists the instructor in the detection and improvement of faulty items in his pool. An elaborate system of preediting of both correct answers and student responses is available. This allows very flexible alterations of "correct" answers and student responses, and permits the assignment of penalty points (e.g., for lateness), or the selective elimination or alteration of specific items, sets, tests, students, etc. We are working on methods of allowing optional and multiple answers to items, weighted appropriately.

ENHANCEMENT OF PERFORMANCE

The CGRT process was conceived and implemented as a production system—to be used repeatedly and routinely by many people. The efficiency and cost of the computing processes are thus important factors in the acceptance of the method both by instructors and by computing centers. Observations of early versions of the test generation program indicated that the development of the test output to be printed was requiring an uncomfortable amount of computer time. (Note that we are discussing central processor time for producing the test output and recording it on some appropriate device such as disk, drum, or tape; the printing of the output is inevitably a lengthy process, but does not place a significant burden on the central processor in modern buffered output systems.) Subsequent timing studies showed that the vast bulk of central processor time was spent in the formatting and outputting processes themselves, with only a small portion of the time spent in the logic of item selection and other computational and input-output activities. Most of the activity in test production is character manipulation of a very elementary kind. The elaborate FORTRAN formatting routines, which were being invoked for each line of output, were inappropriate for such simple but voluminous work. Furthermore, it was felt that too much time was being spent in the library routines which supported the FORTRAN write operation.

As a result of these observations, a version of GENERATOR was prepared which (1) reduced the calls to the FORTRAN formatting routines virtually to zero, and (2) blocked the output internally in the program so as to reduce the calls to the FORTRAN output routines. This was made possible on our Control Data 3600 (later a CDC 6600) equipment by two non-standard but now fairly prevalent FORTRAN features: in-core formatting (specifically, the ENCODE feature), and direct input-output (specifically the BUFFER OUT feature). The former allows data in memory to be manipulated with the customary FORTRAN formatting routines and the result placed in memory, rather than inescapably on an output device. The latter feature permits the writing of an arbitrary amount of data in an arbitrary format onto a file without any editing by the FORTRAN library.

These features were used in the following way: Since the highly repetitive processes in GENERATOR were in the test printing section, all editing of information which would eventually be printed was moved forward in the program and performed (with the aid of the in-core formatting feature) at the earliest practical time. For instance, the text of each item in the pool could be immediately edited and stored to appear as printable lines by inserting the printer control character in the first character position of each line. Other formatting operations, such as formation of question numbers, were moved forward until the number of invocations of a format statement was reduced to one per test! This one involved the first line of the test, which contained the test number. The lines of generated output were collected in a buffer (a FORTRAN array) with appropriate line terminators appended, and under the management of a simple blocking routine were written periodically to the output file.

The results were astonishing, even to seasoned programmers. Table I gives timings for the standard FOR-

TABLE I—Execution Times to Produce 1000 Tests on CDC 6600

	Standard FORTRAN version	Modified CDC 6600 FORTRAN version
Total time in GENERATOR	326 sec.	18 sec.
Time in output section	320	12
Time in item selection section	4	4
Time in other parts of program	2	2

TRAN version of GENERATOR and for our CDC 6600 modified FORTRAN version. The figures are central processor times for preparation of 1000 typical individualized tests, and do not include printing time or time spent waiting for the completion of physical output operations (the latter time is used in a multiprogramming system to run other programs).

One observes an improvement factor of 27 in the crucial output section, with no appreciable cost in other sections of the program! The exact figure will vary with different computer systems, but the obvious conclusion transcends this particular project and this particular machine. Programming language designers and implementers please note. The improvement in performance gained by the above steps changed the CGRT test production operation from one which placed uncomfortable demands on the central processor (which if nothing else usually results in poor turnaround) to one whose cost and performance were quite acceptable. Readers are referred to Reference 1 for a discussion of the economics of the CGRT process.

"MACHINE-INDEPENDENT" FORTRAN PROGRAMS?

The resounding effect of the improvements cited in the previous section has posed a dilemma. One of us (FP) has had considerable experience in developing "machine-independent" FORTRAN programs, and has been an enthusiastic advocate of such coding practices, wherever realistic. Yet the value to the CGRT project of our non-standard practices could not be denied. To compound the problem, there developed considerable interest in the CGRT scheme on the part of teachers and computer people at numerous other locations.

We have developed standard FORTRAN IV versions of GENERATOR and GRADER that are specifically designed to be readily adaptable to most medium and large computer systems. By and large, the syntactic problems in such an effort are minor; virtually everyone has a well-maintained standard FORTRAN compiler at his disposal. However, there are a number of serious semantic difficulties. Although this is not the place to embark on a catalog of possible FORTRAN machine dependencies, several of the problems and our solutions (or lack of them) should be mentioned. The first awkward problem is the packing of characters into words during input and output operations. Different computers have different word sizes, which accommodate various numbers of characters, and the FORTRAN format statements should reflect this fact. Packing is essential in this project, since the item pool frequently contains a large number of characters. Our solution was to provide for each input and output operation a set of read or write and format statements for each common number of characters per word, and to select the proper read/format or write/ format pair by a branch governed by a variable indicating the number of characters per word. The user thus is required to set only a single variable at the beginning of the program.

A second problem is the shifting of characters within words. The usual multiplications or divisions by powers of two inevitably fail on some machine (e.g., the CDC 6600). We reduced the problem in these particular programs to the need to move a character from the leftmost end of a word to the rightmost end. We request the user to supply his own shift routine to accomplish this act, and to replace our routine, which is designed for the CDC 6600.

The generation of acceptable pseudo-random numbers on machines with small word size is difficult to generalize. However, almost every installation will have a random number generator in its library, and we ask that the user replace our generator with a call to his own routine. These are easy and acceptable (if inelegant) solutions to several tedious problems in producing portable FORTRAN programs. There remains the more difficult matter of converting the output in GENERATOR to reduce the dependence on FORTRAN output procedures. With the present inadequate standard FORTRAN, there is no good solution. Our choice was to provide the straightforward FORTRAN program, with copious comments illustrating what one should wish to accomplish with the particular in-core formatting and blocking mechanisms available at his installation.

CGRT WITH SMALL COMPUTERS

CGRT has aroused the interest of teachers in colleges and high schools that have access to small computing systems. We have received many inquiries about the availability of programs for such machines. The IBM 1130 seems to be most widely used in this environment. Therefore we have developed a version of CGRT for a minimal IBM 1130 system equipped with card reader and punch, disk, line printer, and 8K of main memory. At the time of writing, the programs for generating and scoring tests have just completed field test and are ready for distribution.

The tactics employed to implement CGRT on small computers are significantly different from those previously described, although the result is similar. The FORTRAN language available is typically ANSI Basic FORTRAN, a minimal language with few frills. The minimum 8K words of main memory will accommodate on the order of 16,000 characters. Since a modest pool of test items will need over 100,000 characters, the pool must be kept almost entirely on secondary storage. Further, a typical test of three pages will fill main memory, and thus test output must be disposed of promptly. The computer is too small to provide a spooling mechanism (one in which printable material is written to a secondary storage file for later printing), so printing occurs on-line at the time of execution of the program. Since the volume of print is large, and since the typical line printer available on such a system operates at a rather slow speed (80 lines per minute for an IBM 1132), one anticipates that the limiting factor in test generation on the small machine will be the speed of the printer.

On the basis of these indications, our IBM 1130 version of the test generation program was designed to keep the item pool entirely on the disk, using main memory for programs and for organizational data such as the vector of (disk) pointers for each item. Items are selected for inclusion on a test using algorithms similar to those described in an earlier section. The items are then obtained from the disk in the proper order for printing, and are immediately formatted and printed.

The expectation that the test generation process would be limited by the speed of the line printer was confirmed for the IBM 1132 printer. However, on a machine with a 600 line per minute printer, the operation was then limited by the disk access time. We are now developing algorithms to minimize disk accesses so as to again make such faster printers the limiting resource in test generation.

PRESENT STATUS AND PROJECTED DEVELOPMENTS

Computer Generated Repeatable Testing has been a useful adjunct to the instructional process at Indiana University for several years, and more recently at other institutions. A number of uses of repeatable testing have suggested themselves. The principal use has been for testing in college-level classes, frequently but not exclusively in large sections. Repeatable testing in the classroom, at scheduled times outside class, and in a more flexible student-scheduled environment have each proven effective for various instructors. Repeatable testing is well suited for make-up examinations and for administering special tests to allow advanced placement. There are potential applications to correspondence course work, and in regional and national testing centers. Using the repeatable test as a tutorial device, in which the student may take tests simply as a study aid, has become a very important aspect of our service.

For those instructors with available item pools or who are willing to develop items in the necessary quantity, repeatable testing has frequently been a great aid to effective teaching. There are now the beginnings of a coordinated effort to publicize the availability of test items in machine readable form. At the instigation of Gerald Lippey of IBM Corporation (San Jose, California 95114), a meeting was held in San Jose in January 1972 of a small group of people who had worked in the area of mechanized test item banking or computer facilitated testing. As a result of this meeting, Lippey has made a good start toward ascertaining the type and extent of available pools of test items. Through such efforts, and through the generosity of item writers, banks of items may in the future be more readily available than in the past, and each instructor will not be faced with the task of developing his own complete item pool.

At Indiana University, instructors have developed and used repeatable testing item pools in English, geography, home economics, chemistry, economics, statistics, psychology, speech therapy, accounting, education, and others. Usage of CGRT at Indiana University encompasses about ten courses a semester, with over 40,000 individualized tests printed for over 2000 students. Our plans are for expansion of the CGRT facility in several areas. We are working on schemes for student recording (and mechanized recovery) of multi-character answers, to accommodate those instructors who find the single-character answer restriction to be unacceptable. We would like to allow multiple or optional answers to items, with appropriate weights for allocating part credit. There is interest in a more comprehensive item analysis package. And several people are working on the automatic generation of test items.

Repeatable testing programs are available for distribution. The standard FORTRAN version (for medium and large computer systems), the specialized CDC

6600 FORTRAN version, and the IBM 1130 FORTRAN version may be obtained from the authors. If you wish to investigate the CGRT method, please write for documentation and instructions for requesting the programs.

REFERENCE

1 F PROSSER D D JENSEN
Computer generated repeatable tests
Proceedings of 1971 Spring Joint Computer Conference
pp 295–301

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES, INC. (AFIPS)

AFIPS OFFICERS and BOARD OF DIRECTORS

President

Mr. Walter L. Anderson
General Kinetics, Inc.
12300 Parklawn Drive
Rockville, Maryland 20852

Vice President

D. Robert A. Kudlich
Raytheon Co., Equipment Division
Wayland Laboratory
Boston Post Road
Wayland, Massachusetts 01778

Secretary

Mr. Richard B. Blue, Sr.
TRW Systems Group
Scientific Data Processing Lab.
One Space Park—R3/1098
Redondo Beach, California 90278

Treasurer

Mr. George Glaser
McKinsey and Company, Inc.
3000 Sand Hill Road
Menlo Park, California 94025

Executive Director

Dr. Bruce Gilchrist
AFIPS
210 Summit Avenue
Montvale, New Jersey 07645

ACM Directors

Dr. Anthony Ralston
SUNY at Buffalo
Computer Science Department
4226 Ridge Lea Road
Amherst, New York 14226

Mr. Donn B. Parker
Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025

Mr. Herbert S. Bright
Computation Planning, Inc.
5401 Westbard Avenue, Suite 520
Washington, D.C.

IEEE Directors

Dr. A. S. Hoagland
IBM Corporation
Dept. 29A—Building 021
P.O. Box 1900
Boulder, Colorado 80302

Professor Edward J. McCluskey
Stanford University
Department of Electrical Engineering
Palo Alto, California 94305

Dr. S. S. Yau
Department of Electrical Engineering
Stanford University
Palo Alto, California 94305

Simulations Council Director

Mr. Frank C. Rieman
Electronic Associates, Inc.
P.O. Box 7242
Hampton, Virginia 23366

Association for Computation Linguistics Director

Dr. A. Hood Roberts
Center for Applied Linguistics
1717 Massachusetts Avenue, N.W.
Washington, D.C. 20036

American Institute of Aeronautics and Astronautics Director

Mr. Frank Riley, Jr.
Auerbach Corporation
1501 Wilson Boulevard
Arlington, Virginia 22209

American Institute of Certified Public Accounts Director

Mr. Noel Zakin
AICPA
666 Fifth Avenue
New York, New York 10019

American Statistical Association Director

Dr. Mervin E. Muller
5303 Mohican Road
Mohican Hills
Washington, D.C. 20016

American Society for Information Science Director

Mr. Herbert Koller
ASIS
1140 Connecticut Avenue, N.W., Suite 804
Washington, D.C. 20036

Instrument Society of America Director

Mr. Theodore J. Williams
Purdue Laboratory for Applied Industrial Control
Purdue University
Lafayette, Indiana 47907

Society for Industrial and Applied Mathematics Director

Dr. D. L. Thomsen
IBM Corporation
Armonk, New York 10504

Society for Information Display Director

Mr. William Bethke
Rome Air Development Center
RADC (IS, W. Bethke)
Griffiss AFB, New York 13440

Special Libraries Association Director

Mr. Herbert S. White
Institute for Scientific Information
325 Chestnut Street
Philadelphia, Pennsylvania 19105

Association for Educational Data Systems Director

Dr. Sylvia Charp
Director of Instructional Systems
The School District of Philadelphia
Board of Education
5th and Luzerne Streets
Philadelphia, Pennsylvania

JOINT COMPUTER CONFERENCE BOARD

President

Mr. Walter L. Anderson
General Kinetics, Incorporated
12300 Parklawn Drive
Rockville, Maryland 20852

ACM Representative

Dr. Herbert R. J. Grosch
National Bureau of Standards
Center for Computer Science
Washington, D.C. 20234

Vice President

Dr. Robert A. Kudlich
Raytheon Company, Equipment Division
Wayland Laboratory
Boston Post Road
Wayland, Massachusetts 01778

IEEE Representative

Dr. S. S. Yau
Department of Electrical Engineering
The Technological Institute
Northwestern University
Evanston, Illinois 60201

Treasurer

Mr. George Glaser
McKinsey and Company, Iuc.
3000 Sand Hill Road
Menlo Park, California 94025

SCI Representative

Mr. Paul W. Berthiaume
Electronic Associates, Inc.
185 Monmouth Park Highway
West Long Branch, New Jersey 07764

JOINT COMPUTER CONFERENCE COMMITTEE

Mr. Jerry L. Koory, Chairman
H-W Systems
525 South Virgil
Los Angeles, California 90005

JOINT COMPUTER CONFERENCE TECHNICAL PROGRAM COMMITTEE

Mr. Henry S. MacDonald, Chairman
Bell Laboratories
Murray Hill, New Jersey 07971

1973 NATIONAL COMPUTER CONFERENCE CHAIRMAN

Dr. Harvey Garner
Director
Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pennsylvania 17104

1972 FJCC STEERING COMMITTEE

Chairman
Robert Spinrad
Xerox Corporation

Technical Program
Donald A. Meier
National Cash Register

Secretary
Harold Sarkissian
Major Data Corp.

Controller
Howard Verne
Hughes Aircraft Co.

Registration
Patricia Riley
TRW Systems

Local Arrangements
Antonia Schuman
Litton Industries

Printing and Mailing
Katherine Jamerson
Computer Sciences Corp.

Exhibits
A. Luke Ward
San/Bar Electronics Corp.

Public Relations
Allen T. LeAnce
LeAnce and Associates

Special Activities
Fred Gruenberger
San Fernando Valley State College

TECHNICAL PROGRAM COMMITTEE

Chairman
Mr. Donal A. Meier
National Cash Register

Vice-Chairman
Dr. Harold Petersen
RAND Corporation

Speaker Arrangements Director
Mr. Lynn Maxson
IBM Corporation

Liaison & Review Coordinator
Mr. Wolfgang G. Pfeiffer
National Cash Register

Publication Director
Mr. Russell Bennett
Burroughs Corporation

SESSION DIRECTORS

Analysis and Simulation Director
Dr. Ray Nilsen
University of California, Los Angeles

Interdisciplinary Director
Mr. Lowell Amdahl
Compata, Inc.

Software Director
Dr. Richard R. Muntz
University of California, Los Angeles

Users and Applications Director
Mr. Ross F. Penne
University of Southern California

Users and Applications Assoc. Dir.
Dr. Arnold F. Goodman
McDonnell-Douglas Astronautics

Hardware Director
Mr. Jack Pariser
Hughes Aircraft Co.

Systems and Architecture Director
Mr. Harut Barsamian
National Cash Register

SESSION CHAIRMEN, REVIEWERS AND PANELISTS

SESSION CHAIRMEN

Baker, Frank
Balzer, Robert M.
Barsamian, Harut
Bekey, George
Boehm, Barry
Chen, T. C.
Chu, Wesley W.
Denning, Peter J.
Farber, David J.
Fetter, William A.
Flynn, Michael J.
Gaines, Eugene C., Jr.
Gentile, Richard B.
Golub, Eugene
Goodman, Arnold
Hamming, Richard W.
Hollander, G.
Hunter, Kenneth
Husson, Samir
Juncosa, M. L.
Kimbleton, Stephen
Kiviat, Philip J.
Lyon, John K.
McCluskey, E. J.
McManus, Jack
McNamee, Laurence
Mason, Maughn
Mills, Harlan
Mitchell, Gordon
Montgomery, Christine
Morgan, Howard
Newport, Christopher
Patrick, Robert
Penne, Ross F.
Phister, Montgomery, Jr.
Pinkerton, Tad
Reinstedt, Robert
Stefferud, Einar
Taplin, Janet M.
Turn, Rein
Waxman, Ronald
Weissman, Clark
Wilson, Jon C.

REVIEWERS

Alberts, A.
Alrich, J. C.
Anderson, H. M.
Anderson, R.
Arndt, F.
Arnovick, G.
Astrahan, M.
Augustin, D. C.
Avizienis, A.
Ball, N.
Barlow, A. E.
Becker, P.
Bell, T. E.
Bernstein, W. A.
Biener, J. W.
Bloomfield, J.
Boehm, B. W.
Borgsahl, R.
Bork, A.
Branch, R.
Branin, F.
Brereton, T. B.
Brown, A. B.
Calhoon, D.
Canova, G.
Cardwell, D.
Carlson, G.
Carroll, J.
Carter, W. C.
Chen, T. C.
Chernak, J.
Cheydler, B. F.
Choma, J. Jr.
Chu, W. W.
Climenson, W. D.
Copp, D. H.
Courtney, R.
Cowell, W.
Critchlow, A.
Csuri, C.
Dale, A.
Dalrymple, S. H.
Darms, D.
Dittberner, D.
Dorr, F. W.
Duggan, M.
Dumey, A. I.
Eccles, W.
Edwin, L.
Eisenstark, R.
Farmer, N. A.
Feurzeig, W.
Feustel, E. A.
Fiefant, R.
Firschein, O.
Fletcher, J.
Frank, H.
Freilich, A.
Frost, C. R.
Fuches, E.
Fulton, L. M.
Gardner, R.
Gentile, R.
Gillette, G.
Gilliland, B.
Gold, M.
Goodman, A. F.
Gosden, J.
Gotterer, M.
Grandmaison, J.
Grau, A.
Grobstein, D.
Gulick, L. R. Jr.
Hagenstad, M. T.
Hamilton, D. C.
Hammer, C.
Hamming, R. W.
Hammond, F.
Hanson, R. J.
Harper, S.
Harrison, R. L.
Hartwick, R.
Hendrie, G.
Herr, W. B.
Heterick, R. Jr.
Hixon, J.
Hoffman, L.
Hootman, J. T.
Humphrey, R.
Hunt, E.
Hunter, K. W.
Hutt, A. E.
Hyman, M.
Isaksen, L.
Ito, R. A.
Jackson, H. L.
Jeffrey, S.
Jellinek, I.
Jenkins, J. M.
Joseph, E.
Kaltman, A.
Karplus, W. J.

Kay, A.
Keenan, T.
Kernighan, B. W.
Kerr, D. V.
Kimbleton, S. R.
Klein, E.
Kleinrock, L.
Klinger, A.
Klotz, D. A.
Knight, K.
Koory, J.
Kosinski, W.
Kurasch, C.
Kuhns, J. L.
Lange, L.
Larkin, R.
Larson, K.
Lasser, D. J.
Ledley, R.
Leffler, N.
Leichner, G. H.
Levine, L.
Lewis, W.
Lindloon, E.
Linville
Liskov, B.
Loewe, R. T.
Logan, R. S.
Losleben, P.
Luderer, G. W. R.
Lum, V.
Madden, J.
Markel, R.
Marks, H.
Martin, W.
Mathison, S.
Mathur, F.
Mayper, V.
McCracken, D.
McGovern, W.
McIssac, D.
McMurran, M. N.
Meier, D.
Mekota, J.
Mergenweck,
Meuller, M.
Michle, M.
Miller, S.
Miller, W. G.
Mills, H.
Minker, J.
Mitchell
Mittman, B.
Moler, C. B.
Morterana
Myers, R.
Nance, R. E.
Nicols, A. J.
Niedrauer, R. V.
Nielsen, R.
O'Brien, J.
Ofek, H.
Oliver, P.
Onovec
Onyshkevych
Opderbeck, H.
Ostapko, D.
Owens, J.
Pariser, J.
Parker, D.
Patel, A.
Patrick, R. L.
Penne, R.
Petersen, H.
Phillips, T. D.
Pohm, A.
Pomerene, S.
Postel, J.
Price
Prokop, J.
Rajaraman, A. S.
Ramamoorthy, C.
Ray, L.
Reynolds, C.
Rhodes
Rick, J. W.
Rigney, J.
Ripley, G.
Robinson, J.
Robinson, L.
Rodriguez, R.
Rosenbaum, S.
Rosenberg, A. M.
Rosenthal, M.
Rutman, R.
Saal, H.
St. John, D.
Schafer, E.
Schell, R.
Schichman, H.
Schieldge, J.
Schischa, E.
Schneidewind, N.
Schultz, M. H.
Sedelow, W.
Schechter, J.
Short, G. E.
Silvern, L.
Singh, S.
Skelly, P. G.
Small, D. L.
Smith, C.
Smith, R. A.
Southworth, R. W.
Steenbergen, H.
Stefferud, E.
Stephenson, J. W.
Stewart, R. M.
Sturm, W.
Su, S. Y. H.
Summit, R.
Sutherland, W.
Svoboda, A.
Sykes, D.
Taylor, R.
Thomas, R. T.
Tseng, C.
Tucker, S.
Uhlig, R. H.
Uttal, W.
Van Tassel, D.
Walker, D. E.
Watson, R. A.
Watt, W. C.
Weeg, G. P.
Wegbreit, B.
Weiss, E.
Weissman, C.
Werner, J. J. Jr.
Wersan, S.
Whitney, D.
Wiederhold, G.
Wigington, R.
Wiggins
Wilkov, R. S.
Williams, J. G.
Williams, L.
Williams, T. J.
Wilner, W.
Wilson, J.
Wolf, E. W.
Wright, K.
Wyllys, R.
Yakowitz, S.
Yelvington, S.
Young, J.
Zelkovitz, M.

PANELISTS AND SPEAKERS

Donald Aufenkamp, N.S.F.
A. Avizienis, University of Southern California
John Bacon, United California Bank
Max Beere, Tymshare
Barry Boehm, RAND Corporation
Robert Brass, Xerox
Barry Brotman, Allied Chemical Corporation
Gary Carlson, Brigham Young University
Leo Cohen, Consultant
David Copp, Bell Telephone Laboratories
Stephen Crocker, Department of Defense
John Davis, TESDATA Systems Corp.
Lt. Col. Phillip Enslow Jr., Office of Telecommunications Policy, Executive Office of the President
David Evans, Evans and Sutherland
John Farquhar, RAND Corporation
Nick Finamore, Western Electric
H. Fleisher, IBM Corporation
L. Garrett, Motorola
Robert Gordon, Consultant
P. F. Gudenschwager, Honeywell
Richard Hamming, Bell Telephone Laboratories
Cdr. Grace Murray Hopper USNR
Richard Johnson, Stanford University Computation Center
Robert Johnson, Burroughs Corporation
V. Kahan, University of California at Berkeley
Robert Kahn, Bolt, Beranek and Newman, Inc.
E. Mahoney, United States General Accounting Office
C. H. Mays, Fairchild
John McCarthy, Stanford University
M. Douglas McIlroy, Bell Telephone Laboratories
Harry Mergler, Case Western Reserve University
Capt. M. Morris, Federal ADP Simulation Center
Mervin Muller, International Bank for Reconstruction and Development
Peter Newcombe, Brigham Young University
Nils Nilsson, Stanford Research Institute
A. Patel, IBM Corporation
Alan Perlis, Yale University
Charles Perry, McDonnel-Douglas Astronautics
Tom Poole, United Computer Systems
C. Ramamoorthy, University of Texas
Louis Robinson, IBM Corporation
Arthur Rosenberg, Informatics
Capt. Paul Roth, Fleet Combat Direction Systems Support Activity
Stephen Y. Su, University of Southern California
Lee Talbert, Packet Communications, Inc.
L. C. Thomas, Bell Telephone Laboratories
D. E. Walker, S.R.I.
P. Weber, Lane County
Mark Wells, Los Alamos Scientific Laboratory
James Williams, United States General Accounting Office
Joe Wineke, Compress, Inc.
M. Worthy, Operating Systems
Gordon Zeller, *Los Angeles Times*

PRELIMINARY LIST OF EXHIBITORS

Addison-Wesley Publishing Company, Inc.
Addmaster Corporation
Addressograph Multigraph Corporation
AFIPS Press
American Elsevier Publishing Company
American Telephone & Telegraph
Ampex Corporation
Anaheim Publishing Company
Ansul Company
Basic Timesharing, Inc.
Beehive Terminal
Bridge Data Products, Inc.
Burroughs Corporation
Caelus Memories, Inc.
Centronics
Century Electronics and Instruments
Cipher Data Products
Codex Corporation
Collins Radio Company
ComData Corporation
Computer Access Systems, Inc.
Computer Automation, Inc.
Computer Copies Corporation
Computer Design Publishing Corporation
Computer Machinery Corporation
Controls Research Corporation
Courier Terminal Systems, Inc.
Data Disc, Inc.
Data General Corporation
Datamation
Data Printer Corporation
Datapro Research Corporation
Data Products Corporation
Dataram Corporation
Datawest Corporation
Datum, Inc.
Diablo Systems, Inc.
Digital Computer Controls, Inc.
Digital Development Corporation
Documation, Inc.
DuPont Company
Eastman Kodak Company
Electronic Engineering Company of California
Electronic News, Fairchild Publications
Facit-Odhner, Inc.
Federal Screw Works
Floating Point Systems, Inc.
General Automation, Inc.
GTE Lenkurt
G-V Controls
Hayden Publishing Company, Inc.
Hewlett-Packard Company
Honeywell Computer Journal
Houston Instrument
IMSL
Inforex, Inc.
Information Data Systems, Inc.
Infosystems
Infoton, Inc.
Intel Corporation
International Communications Corporation
International Computer Products, Inc.
Kennedy Company
Kybe Corporation
Lipps, Inc.
Litton ABS OEM Products
Lorain Products Corporation
Marubeni America Corporation
Microdata Corporation
Micro Switch
Milgo Electronic Corporation
Miratel Division—Ball Brothers Research Corp.
Modern Data
Mohawk Data Sciences Corporation
Northern Electric Company, Ltd.
Nortronics Company, Inc.
Olympia USA, Inc.
Ovonic Memories, Inc.
Panasonic
Paradyne Corporation
Pertec Corporation
Pioneer Electronics Corporation
Pioneer Magnetics, Inc.
Potter Instrument Company, Inc.
Prentice Hall, Inc.
Printer Technology, Inc.
Producers Service Corporation
Radley Associates Limited
Randomex, Inc.
Raymond Engineering, Inc.
Raytheon Service Company
Redactron Corporation

Remex, A unit of Ex-Cell-O Corporation
Sangama Electric Company
Signal Galaxies, Inc.
The Singer Company
Sycor, Inc.
Sykes
Systems Furniture Company
Tally Corporation
Techtran Industries, Inc.
Tekronix, Inc.
Tele-Dynamics
Teleprocessing Industries, Inc.
Teletype Corporation
Texas Instruments, Inc.
Toko, Inc.
Tri-Data Corporation
Van San Corporation
Vector General, Inc.
Velo-Bind
Wangco, Inc.
John Wiley and Sons, Inc.
Xerox Corporation

AUTHOR INDEX