HEWLETT-PACKARD
JOURNAL

table of contents

June 1994,
Volume 45, Issue 3

Articles

1 **Corporate Business Servers: An Alternative to Mainframes for Business Computing**
by Thomas B Alexander, Kenneth G. Robertson, Dean T. Lindsay, Donald L. Rogers, John R. Obermeyer, John R. Keller, Keith Y. Oka, and Marlin M. Jones, II

2 **PA-RISC Symmetric Multiprocessing in Midrange Servers**
by Kirk M. Bresniker

3 **SoftBench Message Connector: Customizing Software Development Tool Interactions**
by Joseph J. Courant

4 **Six-Sigma Software Using Cleanroom Software Engineering Techniques**
by Grant E. Head

5 **Fuzzy Family Setup Assignment and Machine Balancing**
by Jan Krucky

Corporate Business Servers: An Alternative to Mainframes for Business Computing

With expandable hardware, PA-RISC architecture, symmetric multiprocessing, a new bus structure, and robust error handling, these systems provide a wide range of performance and configurability within a single cabinet. Standard features include one to twelve symmetric PA-RISC 7100 multiprocessors optimized for commercial workloads, main memory configurations from 128M bytes to 2G bytes, and disk storage up to a maximum of 1.9 terabytes.

by Thomas B. Alexander, Kenneth G. Robertson, Dean T. Lindsay, Donald L. Rogers, John R. Obermeyer, John R. Keller, Keith Y. Oka, and Marlin M. Jones, II

The overall design objective for the HP 9000 Model T500 corporate business server (Fig. 1) was to set new standards for commercial systems performance and affordability. Combining expandable hardware, PA-RISC architecture, symmetric multiprocessing with up to 12 processors, a new bus design, robust error handling, and the HP-UX* operating system, the Model T500 delivers a cost-effective alternative to mainframe solutions for business computing.

Users of HP's proprietary operating system, MPE/iX, also enjoy the benefits of the Model T500 hardware. These systems are designated the HP 3000 Series 991/995 corporate business systems. They provide high performance by supporting from one to eight processors with superior value for their class. The MPE/iX system is designed to support business-critical data and offers features such as powerful system management utilities and tools for performance measurement.

In this paper, the hardware platform for both the HP-UX and the MPE/iX systems will be referred to as the Model T500. The Model T500 is an update of the earlier HP 9000 Model 890/100 to 890/400 systems, which supported from one to four PA-RISC processors operating at 60 MHz. For MPE/iX, the Series 991/995 is an update of the earlier Series 990/992 systems.

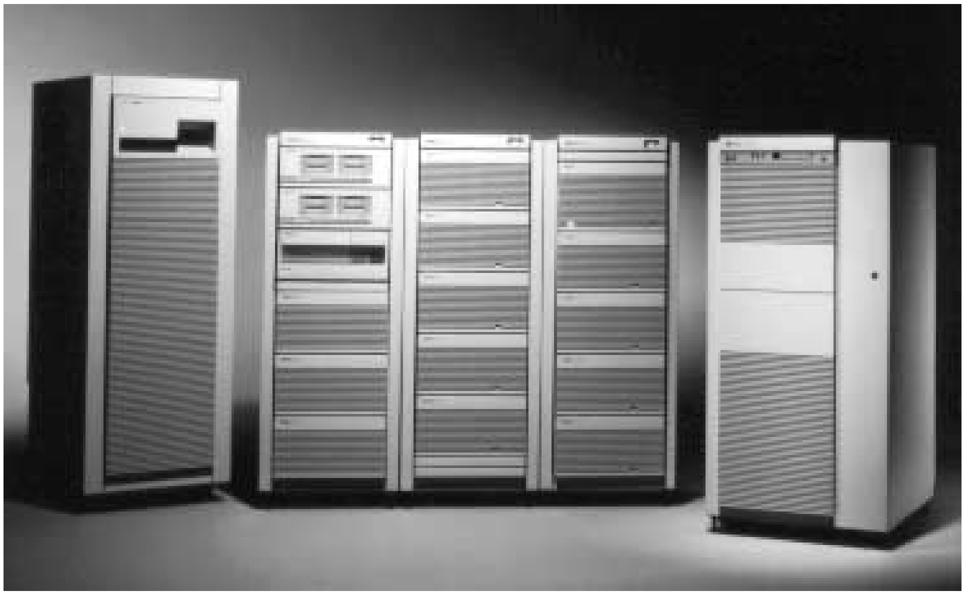


Fig. 1. The HP 9000 Model T500 corporate business server (right) is designed as an alternative to mainframe solutions for online transaction processing and other business computing applications. It runs the HP-UX operating system. The same hardware running the MPE/iX operating system is designated the HP 3000 Series 991/995 corporate business systems. The Model T500 SPU (right) is shown here with various peripherals and expansion modules.

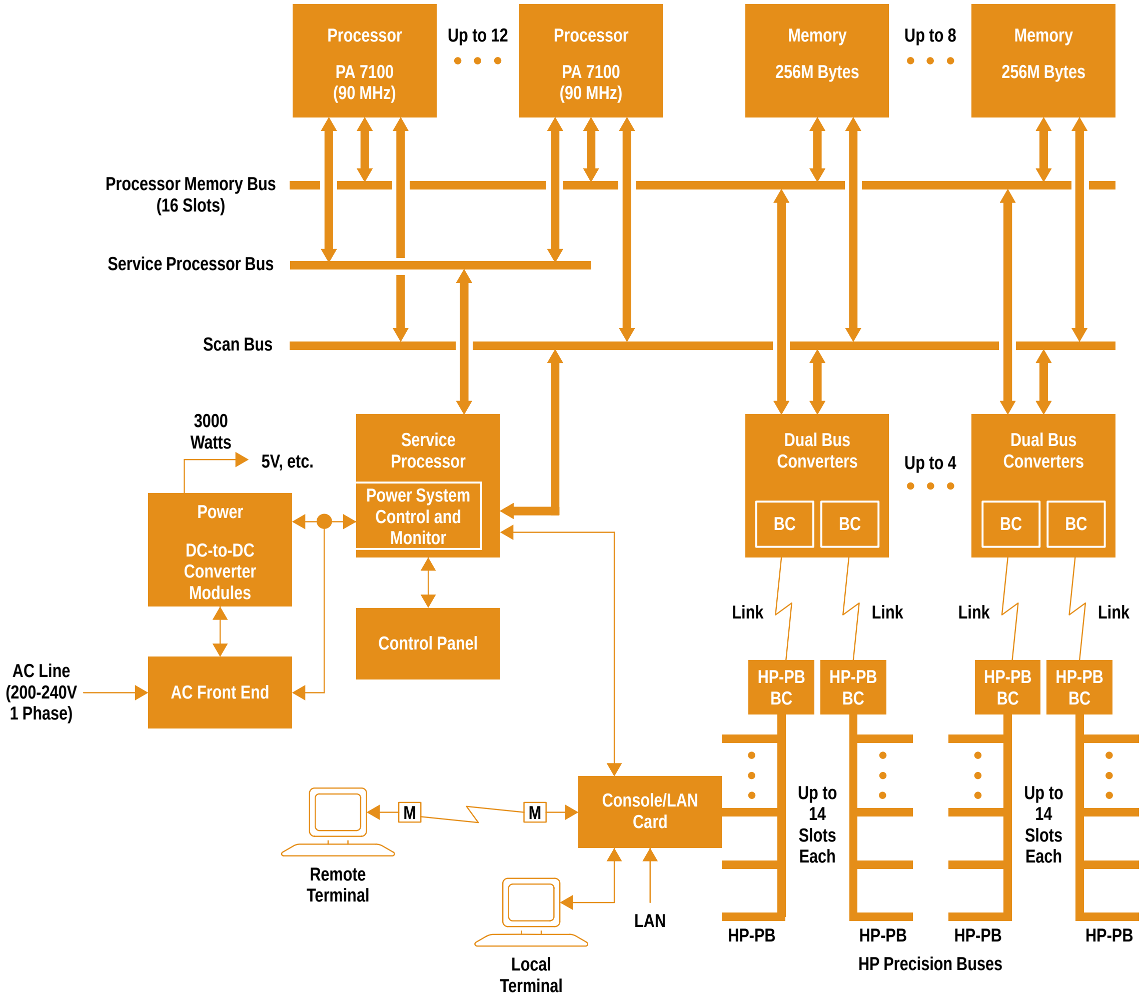


Fig. 2. HP 9000 Model T500 system processing unit block diagram.

Standard features of the Model T500 include one to twelve symmetric PA-RISC multiprocessors optimized for commercial workloads and operating at 90 MHz, main memory configurations from 128M bytes to 2G bytes,† and disk storage up to a maximum of 1.9 Tbytes (1900 Gbytes). This expandability allows the Model T500 to provide a wide range of performance and configurability within a single cabinet. The Model T500's package minimizes the required floor space, while air cooling removes the need for expensive mainframe-type cooling systems.

The Model T500 is designed to provide leading price/performance. The HP 9000 Model T500 with six PA-RISC 7100 processors operating at 90 MHz has achieved 2110.5 transactions per minute on the TPC-C benchmark (U.S.$2115 per tpmC).†† The SPECrate (SPECrate_int92 and SPECrate_fp92) benchmark results show linear scaling with the number of processors, which is expected for CPU-intensive workloads with no mutual data dependencies. The Model T500/400 reaches 38,780 SPECrate_fp92 and 23,717 SPECrate_int92 with twelve processors.

The Model T500 provides this high level of performance by using a balanced bus architecture. The processor memory bus currently provides the main processor-to-memory or processor-to-I/O interconnect with a bandwidth of 500 Mbytes/s and a potential capability up to 1 Gbyte/s. The I/O buses provide a total aggregate I/O bandwidth of 256 Mbytes/s. These bandwidths satisfy the high data sharing requirements of commercial workloads.

System Overview

The key to the Model T500's expandability and performance is its bus structure. The processor memory bus provides a high-bandwidth coherent framework that ties the tightly coupled symmetrical multiprocessing PA-RISC processors together with I/O and memory. Fig. 2 shows a block diagram of the Model T500.

The processor memory bus is a 60-MHz bus implemented on a 16-slot backplane with eight slots suitable for processors or memory boards and eight slots suitable for I/O adapters or memory boards. Each slot can contain as many as four modules, and can obtain its fair fraction of the bandwidth provided by the system bus. Custom circuit designs allow the bus to operate at a high frequency without sacrificing physical

† Hewlett-Packard Journal memory size conventions:

1 kbyte = 1,000 bytes	1K bytes = 1,024 bytes
1 Mbyte = 1,000,000 bytes	1M bytes = 1,024^2 bytes = 1,048,576 bytes
1 Gbyte = 1,000,000,000 bytes	1G bytes = 1,024^3 bytes = 1,073,741,824 bytes
1 Tbyte = 1,000,000,000,000 bytes	

†† The Transaction Processing Council requires that the cost per tpm be stated as part of the TPC performance results. Cost per tpm will vary from country to country. The cost stated here is for the U.S.A.

connectivity. To prevent system bus bandwidth from becoming a bottleneck as the number of processors increases, the bus protocol minimizes bus contention and unproductive traffic without adding undue complexity to the bus modules.

To support such a large number of slots the processor memory bus is physically large and has an electrical length of 13 inches. State-of-the art VLSI design and mechanical layout allow the processor memory bus to run at 60 MHz—a very high frequency of operation for a bus of this size.

The input/output subsystem links the processors and memory on the processor memory bus to I/O devices, including a variety of networks. The Model T500 supports attachment of up to eight Hewlett-Packard precision buses (HP-PB), each of which connects up to 14 I/O cards. The first HP-PB is internal to the Model T500 and the other HP-PBs would be located in adjacent racks. Each HP-PB connects to the processor memory bus through a linked bus converter consisting of a dual bus converter, a bus converter link, and an HP-PB bus converter. Under normal operating conditions the bus converters are transparent to software.

The service processor consists of a single card whose purpose is to provide hardware control and monitoring functions for the Model T500 and a user interface to these functions. To achieve this purpose, the service processor has connectivity to many parts of the Model T500. The scan bus is controlled by the service processor and provides the service processor with scan access to all of the processor memory bus modules. The scan bus is used for configuration of processor memory bus modules and for manufacturing test. The service processor also provides the clocks used by processor memory bus modules and controls the operation of these clocks. The service processor provides data and instructions for the processors over the service processor bus during system initialization and error recovery. The service processor connects to the control panel and provides the system indications displayed there. The service processor provides its user interface on the console terminals through its connection to the console/LAN card.

The service processor also contains the power system control and monitor, which is responsible for controlling and monitoring the Model T500's power and environmental system. The main power system receives 200-240V single-phase mains ac and converts it to 300Vdc. This 300V supply is then converted by various dc-to-dc converter modules to the needed system voltages (e.g., one module is 300Vdc to 5Vdc at 650W.). The power system control and monitor additionally controls the system fans and power-on signals. The power system control and monitor performs its functions under service processor control and reports its results to the service processor.

Processor Memory Bus

The present implementation of the Model T500 uses 90-MHz PA-RISC central processing units (CPUs)[1,2,3] interconnected with a high-speed processor memory bus to support symmetric twelve-way multiprocessing. This section focuses on the features and design decisions of the processor memory bus, which allows the system to achieve excellent online transaction processing (OLTP) performance and efficient multiprocessor scaling.

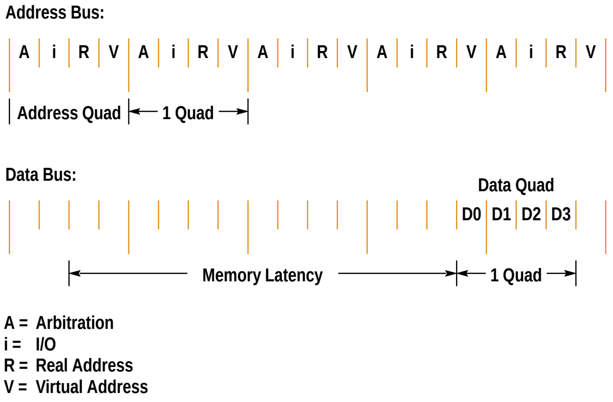


Fig. 3. Processor memory bus pipeline.

Bus Protocol

The processor memory bus is a synchronous pipelined bus. The pipelined nature of the bus protocol places it between a split transaction protocol and an atomic transaction protocol. This allows the processor memory bus to have the performance of a split transaction bus with the lower implementation complexity of an atomic transaction bus.

The processor memory bus has separate address and data buses. The address bus is used to transfer address and control information and to initiate transactions. Non-DMA I/O data is also transferred on the address bus. The data bus transfers memory data in blocks of 16, 32, or 64 bytes. The processor data bus in the present implementation is 64 bits wide, although the protocol, backplane, and memory system also support 128-bit-wide accesses. For 32-byte transfers on the data bus, the available bandwidth is 480 Mbytes per second. If processors use all 128 bits of the data bus to perform 64 byte transfers, the bandwidth doubles to 960 Mbytes per second.

Fig. 3 shows the processor memory bus pipeline. Four consecutive processor memory bus states are referred to as a *quad*. A transaction consists of a quad on the address bus, followed at some fixed time by a quad on the data bus.

An address quad consists of an arbitration cycle, an I/O cycle, a real address cycle, and a virtual address cycle. The arbitration cycle is used by bus masters to arbitrate for use of the bus. The I/O cycle is used to transfer data in the I/O address space. The real address cycle is used to transfer the memory or I/O address and to indicate the transaction type. The virtual address cycle is used to transfer the virtual index for cache coherency checks.

A data quad consists of four data transfer cycles. The fixed time between address and data quads is programmed at system initialization. This arrangement allows multiple pipelined transactions to be in progress at the same time. Since data is returned at a fixed time after the address quad, the module returning data automatically gets access to the data bus at that time. The set of supported transactions includes reads and writes to memory address space, reads and writes to I/O address space, and cache and TLB (translation lookaside buffer) control transactions.

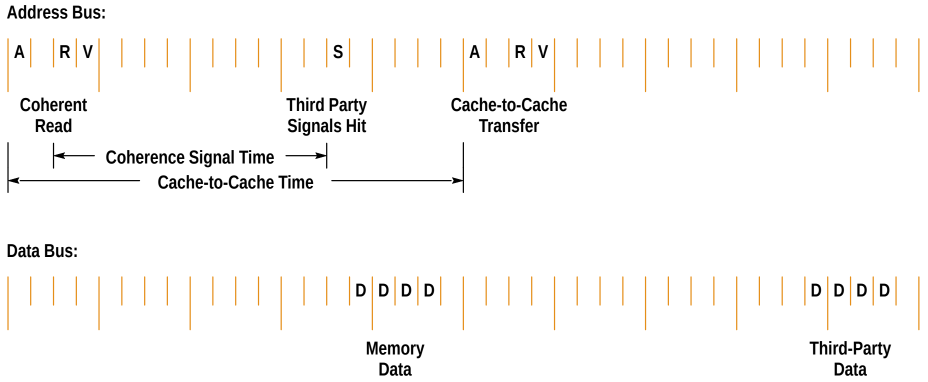


Fig. 4. Multiprocessor protocol, showing cache-to-cache transfer timing.

If a transaction is initiated on the processor memory bus, but a module (either the slave or a third party) is not prepared to participate in the transaction, that module has the option of *busying* the transaction. When the master sees that its transaction is busied, it must retry the transaction at a later time. Busy is appropriate, for example, when the bus adapter is asked to forward a read transaction to the lower-speed precision bus (see "Arbitration" below for more information).

For cases in which a module requires a brief respite from participating in transactions, it can *wait* the bus, that is, it can freeze all address and data bus activity. It does this by asserting the wait signal on the bus. The wait facility is analogous to a stall in the processor pipeline.

Multiprocessor Bus Protocol

The processor memory bus provides cache and TLB coherence with a *snoopy*[4] protocol. Whenever a coherent transaction is issued on the bus, each processor (acting as a third party) performs a cache coherency check using the virtual index and real address.

Each third-party processor is responsible for signaling cache coherency status at a fixed time after the address quad. The third party signals that the cache line is in one of four states: shared, private clean, private dirty, or not present. The requesting processor interprets the coherency status to determine how to mark the cache line state (private clean, private dirty, or shared). The third party also updates its cache line state (no change, shared, or not present).

If a third party signals that it has the requested line in the private dirty state, then it initiates a cache-to-cache transaction at a fixed time after the address quad. The requesting processor discards the data received from main memory for the initial request and instead accepts the data directly from the third party in a cache-to-cache transfer. At this same time the data from the third party is written to main memory. The timing of these events is shown in Fig. 4.

Since the processor memory bus allows multiple outstanding pipelined transactions, it is important that processor modules be able to perform pipelined cache coherency checks to take maximum advantage of the bus bandwidth. Fig. 5 shows an example of pipelined cache coherency checking.

Programmable Parameters

The processor memory bus protocol permits many key bus timing parameters to be programmed by initialization software. Programming allows different implementations to optimize the parameter values to increase system performance and reduce implementation complexity. Initialization software calculates the minimum timing allowed for the given set of installed bus modules. As new modules are designed that can operate with smaller values (higher performance), initialization software simply reassigns the values.

The programmable parameters include:

- Address-to-Data Latency. The time from the real address of the address quad to the first data cycle of the data quad. The present implementation achieves a latency of 217 nanoseconds.
- Coherency Signaling Time. The time required for a processor to perform a cache coherency check and signal the results on the bus.
- Cache-to-Cache Time. The time from the address quad of the coherent read transaction to the address quad of the cache-to-cache transaction. This value is the time required

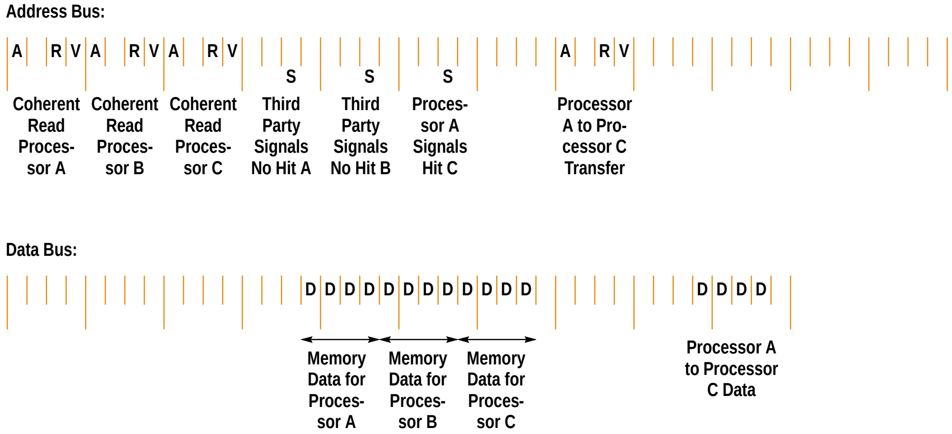

Address Bus:
A
R V A
R V A
R V
S
S
S
A
R V
Coherent Read Processor A
Coherent Read Processor B
Coherent Read Processor C
Third Party Signals No Hit A
Third Party Signals No Hit B
Processor A Signals Hit C
Processor A to Processor C Transfer
Data Bus:
D D D D D D D D D D D D
D D D D
Memory Data for Processor A
Memory Data for Processor B
Memory Data for Processor C
Processor A to Processor C Data

Fig. 5. Pipelined cache coherency checking.

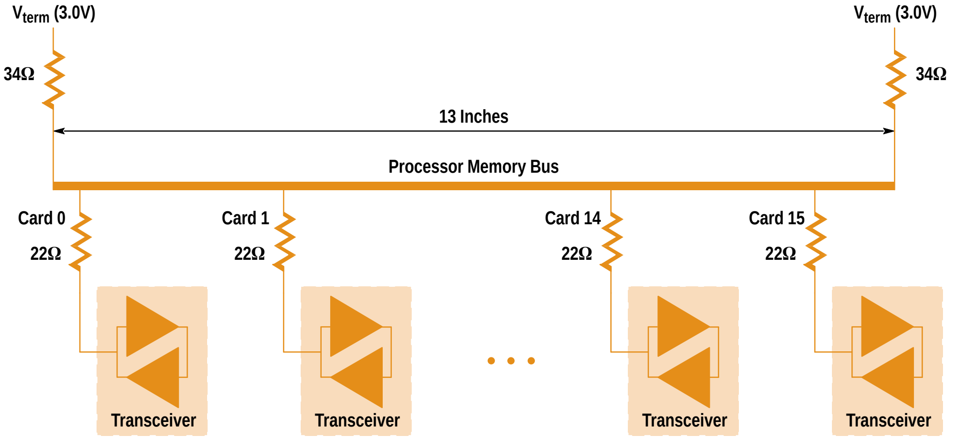


Fig. 6. Processor memory bus electrical layout.

for a processor to do a cache coherency check and copy out dirty data.

- Memory Block Recovery Time. The time it takes a memory block to recover from an access and become ready for the next access.
- Memory Interleaving. The memory block identifier assignments. The assignments depend on the size and number of memory blocks installed in the system.

Arbitration

The processor memory bus uses three different arbitration rules to determine when a module can get access to the bus. The first rule, used for references to memory, states that a master can arbitrate for a memory block only after the block has recovered from the previous access. Bus masters implement this by observing all transactions on the processor memory bus. Since memory references include the block identifier and the recovery times are known, masters refrain from arbitration for a busy block. The benefit of this arbitration rule is that memory modules do not have to queue or busy transactions, and therefore bus bandwidth is conserved because every memory transaction is a useful one.

The second arbitration rule, used for references to I/O address space, requires that a master of a busied I/O transaction not retry the transaction until the slave has indicated that it is ready to accept the transaction. The slave indicates readiness by asserting the original master's arbitration bit on the bus. The master detects that the slave has restarted arbitration and continues to attempt to win arbitration. This rule prevents masters from wasting bus bandwidth by continually retrying the transaction while the slave is not ready to accept it, and avoids most of the complexity of requiring slaves to master a return transaction.

The third mechanism, referred to as *distributed priority list* arbitration, is invoked when multiple masters simultaneously arbitrate for the processor memory bus. Distributed priority list arbitration is a new scheme for general arbitration. It uses a least-recently-used algorithm to determine priority on the bus. A master implements distributed priority list arbitration by maintaining a list of masters that have higher priority than itself and a list of masters that have lower priority. Thus, an arbitrating master can determine it has won by observing that no higher-priority master has arbitrated. The identity of the winning master is driven onto the processor memory bus in the address quad. Masters then update their lists to indicate they now have higher priority than the winner. The winner becomes the lowest priority on all lists. This arbitration scheme guarantees fair access to the bus by all masters.

Electrical Design

The processor memory bus has the somewhat conflicting goals of high connectivity (which implies a long bus length) and high bandwidth (which implies a high frequency of operation and a correspondingly short bus length). A typical solution to these goals might use custom transceivers and operate at a frequency of 40 MHz. However, by using custom VLSI, incident wave switching, and state-of-the-art design, the Model T500 processor memory bus allows reliable operation at 60 MHz over a 13-inch bus with 16 cards installed.

Each board on the processor memory bus uses two types of custom bus interface transceiver ICs. The first IC type incorporates 10 bits of the processor memory bus (per package), error detection and correction logic, and two input ports (with an internal 2:1 multiplexer) in one 100-pin quad flat package. This IC is referred to as a processor memory bus transceiver in this article. The second IC type performs all of the above duties but adds arbitration control logic and control of 20 bits on the processor memory bus in a 160-pin quad flatpack. This IC is referred to as an arbitration and address buffer in this article. The arbitration and address buffer and the processor memory bus transceivers are implemented in HP's 0.8-micrometer CMOS process.

Fig. 6 shows the basic processor memory bus design. Each processor memory bus signal line has a 34-ohm termination resistor tied to 3V at each end of the bus. Each card installed on the processor memory bus has a series terminating resistor of 22 ohms between the connector and a corresponding bidirectional buffer transceiver for the processor memory bus.

For asserted signals (active low) the output driver transistor in Fig. 7 turns on. This pulls the 22-ohm resistor to approximately ground which (through the resistor divider of 22 ohms and two 34-ohm resistors in parallel) pulls the processor memory bus signal to approximately 1.6 volts. On deasserted signals the output driver is off and the 34-ohm resistors at each end of the bus pull the bus to a high level of approximately 3V.

The receiver (a greatly simplified version is shown in Fig. 7) is a modified differential pair. One input of the differential pair is connected to an external reference voltage of 2.55V. The other input of the differential pair is connected to the

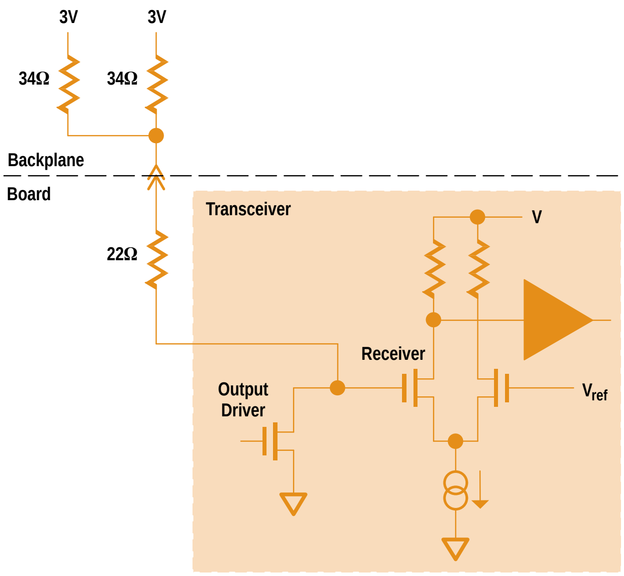


Fig. 7. Processor memory bus electrical detail.

processor memory bus. Use of a differential pair receiver allows incident signal switching (i.e., the first transition of the signal is detected by the receiver) and precise level control of the input switch point.

The 22-ohm series resistor performs several important functions. First, when a transceiver asserts a signal, the resistor limits pull-down current. Second, for boards where the transceiver is not driving, the 22-ohm resistor helps isolate the processor memory bus from the capacitive and inductive load presented by the inactive buffers and board traces. Lastly, the 22-ohm resistor helps dampen transient waveforms caused by ringing.

Processor Board

The processor board used in the Model T500 is a hardware performance upgrade product that replaces the original processor board of the HP 9000 Model 890 corporate business server. With up to two processor modules per processor board, the dual processor board allows the Model T500 system to achieve up to twelve processor systems. Additionally, the use of the PA-RISC 7100 processor improves uniprocessor performance. The key features of the Model T500's processor include:

- Direct replacement of the original processor board (cannot be mixed with original processor boards in the same system).
- Increased multiprocessing performance with support for one to twelve CPUs.
- Processor modules based on 90-MHz PA-RISC 7100 CPU chip[8] with on-chip floating-point coprocessor for higher uniprocessor integer and floating-point performance.
- Processor modules that allow single-processor and dual-processor configurations per processor slot. Easy field upgrade to add a second processor module to a single processor module board.
- Processor clock frequency 90 MHz, processor memory bus clock frequency 60 MHz.
- 1M-byte instruction cache (I cache) and 1M-byte data cache (D cache) per module.

Performance Improvement

Relative to its predecessor, the Model T500's processor board SPEC integer rate is improved by a factor of 1.9 times and the SPEC floating-point rate is improved by a factor of 3.4 times. The Model T500's processor performance relative to its predecessor is shown in the table below.

	Model T500	**Model 890**
CPU Clock	90 MHz	60 MHz
Bus Clock	60 MHz	60 MHz
Cache Size (per CPU)	1M-byte I cache, 1M-byte D cache (direct mapped)	2M-byte I cache, 2M-byte D cache (2-way set-associative)
SPECint92	98.3	Not Published
SPECfp92	170.2	Not Published
SPECrate_int92 (1 CPU)	2310	1215
SPECrate_int92 (2 CPUs)	4609	2253
SPECrate_int92 (4 CPUs)	9017	4301
SPECrate_int92 (8 CPUs)	17114	N/A
SPECrate_int92 (12 CPUs)	23717	N/A
SPECrate_fp92 (1 CPU)	4019	1180
SPECrate_fp92 (2 CPUs)	7963	2360
SPECrate_fp92 (4 CPUs)	15341	4685
SPECrate_fp92 (8 CPUs)	28341	N/A
SPECrate_fp92 (12 CPUs)	38780	N/A
tpsA	Not Published	710.43 tpsA at U.S.$8,258 per tpsA
tpmC	2110.5 tpmC at U.S.$2,115 per tpmC	Not Published

Hardware Overview

The Model T500's processor board consists of one or two processor modules, a set of 12 processor memory bus transceivers (4 address and 8 data bus transceivers), an arbitration and address buffer, two processor interface chips, two sets of duplicate tag SRAMs, ECL clock generation circuitry, four on-card voltage regulators, scan logic circuitry, connectors, a printed circuit board, and mechanical hardware. Fig. 8 shows the processor board hardware block diagram. Fig. 9 is a photograph of a processor board with two processor modules.

Processor Modules

The processor board is centered around two identical, removable processor modules based on the HP PA 7100 CPU chip. Each module consists of a CPU chip, 26 SRAMs which make up the 1M-byte instruction cache (I cache) and 1M-byte data cache (D cache), a 4.1-inch-by-4.4-inch 12-layer printed circuit board, and a 100-pin P-bus connector.

Each processor module communicates with its processor interface chip through a 60-MHz, 32-bit multiplexed address/data bus called the P-bus. Each module has a dedicated P-bus. The P-bus has 35 data and address lines and 18 control lines.

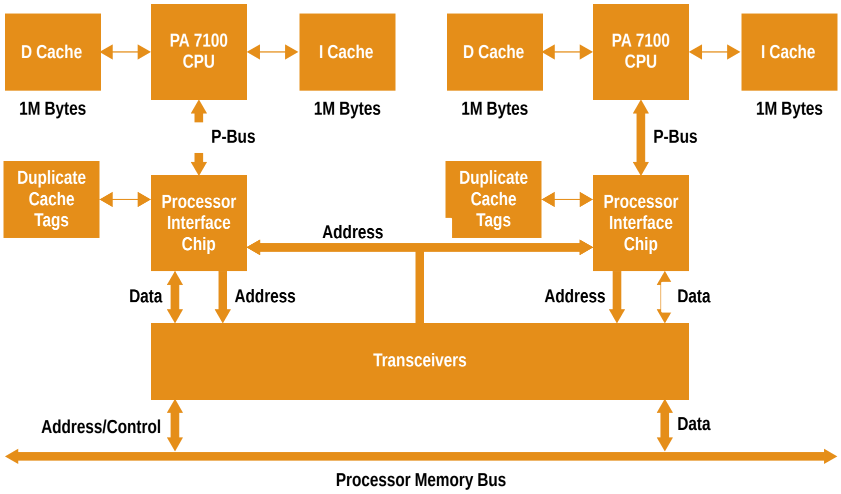


Fig. 8. Processor board organization.

The I cache and D cache each have the following features:

- 64-bit access (I cache 64-bit double-wide word, D cache two 32-bit words)
- Direct mapped with a hashed address and virtual index
- Bandwidth up to 520 Mbytes/s
- I and D cache bypassing
- Parity error detection in both I and D caches (parity errors in the I cache cause a refetch of the offending instruction)
- 32-byte cache line size.

The CPU chip has the following features:

- Level 1 PA-RISC implementation with 48-bit virtual addressing
- Addresses up to 3.75G bytes of physical memory
- Multiprocessor cache coherency support
- TLB (translation lookaside buffer)
  - 120-entry unified instruction and data TLB
  - Fully associative with NUR (not used recently) replacement
  - 4K page size
- Floating-point coprocessor
  - Located on-chip
  - Superscalar operation
  - Multiply, divide, square root
  - Floating-point arithmetic logic unit (FALU)
- P-bus system interface (to bus interface chip)

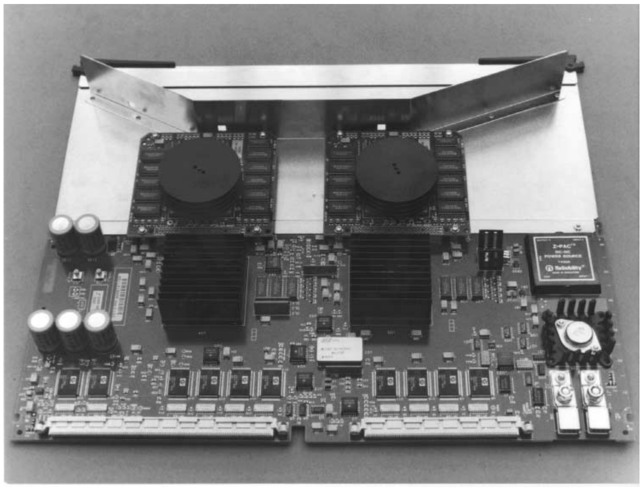


Fig. 9. Model T500 processor board with two processor modules.

- Serial scan path for test and debug
- Operation from dc to 90 MHz
- Performance improvements
  - Load and clear optimizations
  - Hardware TLB miss handler support
  - Hardware static branch prediction
- 504-pin interstitial pin-grid array package.

Processor Interface Chip

Each processor interface chip transmits transactions between its CPU and the rest of the system (memory, I/O, and other processors) via the processor memory bus. The processor interface chip for each processor module interfaces its CPU (through the P-bus) to the the processor memory bus transceivers and the arbitration and address buffer. The CPU's line size is 32 bytes, so the processor interface chip provides a 64-bit data interface to the processor memory bus transceivers. The two processor interface chips communicate through separate ports on the processor memory bus transceivers, which provide the required multiplexing internally.

Each processor interface chip also contains an interface that allows it to communicate with self-test, processor dependent code (boot and error code), and processor dependent hardware (time-of-day clock, etc.) on the service processor board. The processor interface chip is implemented in HP's 0.8-micrometer CMOS process and is housed in a 408-pin pin-grid array package.

The processor interface chip has two features to enhance the multiprocessor performance of the system: duplicate data cache tags and coherent write buffers. The coherent buffers support the processor memory bus's multiprocessor implementation of cache coherence protocol.

Duplicate Data Cache Tags. The interface chip maintains its own duplicate copy of the CPU's data cache tags in off-chip SRAMs. The tags contain the real address of each cache line and the valid and private bits (but not the dirty bit). The duplicate cache tags are kept consistent with the CPU's data cache tags based only on the transactions through the interface chip. The duplicate tags allow the interface chip to signal the status of a cache line during a coherent transaction without querying the processor (which would require a pair of transactions on the P-bus). Measurements (using the processor interface chip's built-in performance counters) for a

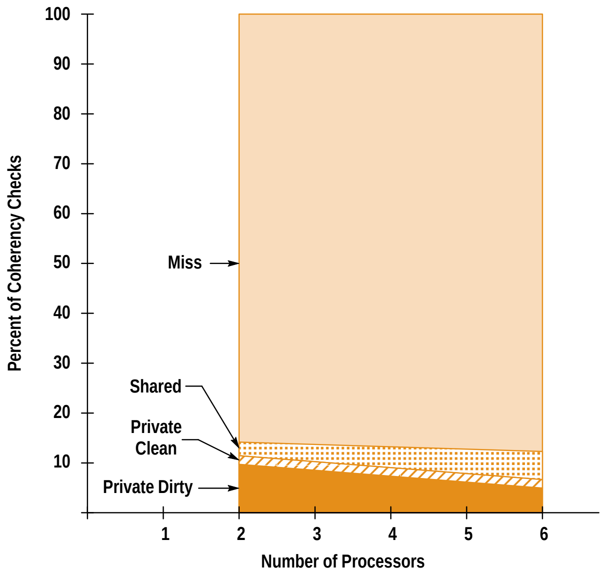


Fig. 10. Data sharing in a multiprocessor Model 890 system as measured by results of cache coherency checks.

wide variety of benchmarks show that for 80 to 90 percent of coherent transactions, the cache line is not present in a third-party CPU's data cache, as shown in Fig. 10. The duplicate tags increase system performance to varying degrees for different workloads. Measurements on a four-processor system show duplicate tags increase system throughput by 8% for a CPU-intensive workload and 21% for a multitasking workload.

Coherent Write Buffers. To isolate the CPU from traffic on the bus, the interface chip contains a set of five cache line write buffers. The buffers are arranged as a circular FIFO memory with random access. If the CPU writes a line to memory, the interface chip stores the line in one of its buffers until it can win arbitration to write the line to memory. While the line is in a buffer, it is considered part of the CPU's cached data from the system bus point of view and participates in coherence checking on the bus. These buffers are also used for temporary storage of data sent from the cache as a result of a coherency check that hits a dirty cache line. By having many buffers, the interface chip is able to handle multiple outstanding coherency checks.

Pipeline

The PA 7100 pipeline is a five-stage pipeline. One and a half stages are associated with instruction fetching and three and a half stages are associated with instruction execution. The PA 7100 also has the ability to issue and execute floating-point instructions in parallel with integer instructions. Fig. 11 shows the CPU pipeline.

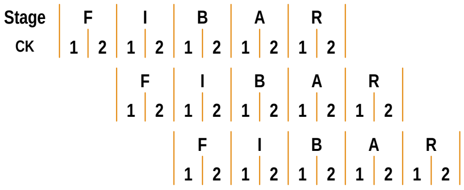


Fig. 11. CPU pipeline.

Instruction fetch starts in CK1 of stage F and ends in CK1 of stage I. For branch prediction, the branch address is calculated in CK1 of I and completes by the end of CK2 of I. This address is issued to the I cache.

From CK2 of I to CK1 of B, the instruction is decoded, operands are fetched, and the ALU and SMU (shift merge unit) produce their results. The data cache address is generated by the ALU by the end of CK1 of B. For branch prediction, the branch address is calculated in CK1 of B and completes by the end of CK2 of B.

Data cache reads start in CK2 of B and end in CK2 of A. Load instructions and subword store instructions read the data portion of the D cache during this stage. For all load and store instructions the tag portion of the D cache is read during this stage. The tag portion of the D cache is addressed independently from the data portion of the D cache so that tag reads can occur concurrently with a data write for the last store instruction. Branch condition evaluation is completed by the end of CK2 of B.

The PA 7100 CPU maintains a store buffer which is set on the cycle after CK2 of A of each store (often CK2 of R). General registers are set in CK2 of R. The store buffer can be written to the D cache starting on CK2 of R and continuing for a total of two cycles. The store buffer is only written on CK2 of R when one of the next instructions is a store instruction. Whenever the next store instruction is encountered, the store buffer will be written out to the cache.

Clock Generation

The clock generation circuitry provides 60-MHz and 90-MHz differential clock signals to the processor memory bus interface ports and the processor modules, respectively. The Model T500's processor board uses a hybrid phase-locked loop component developed especially for the Model T500. The phase-locked loop generates a synchronized 90-MHz processor clock signal from the 60-MHz processor memory bus clock. Clock distribution is by differential ECL buffers with supplies of +2.0V and −2.5V. The use of offset supplies for the ECL allows optimal termination with the 50-ohm termination resistors tied directly to ground, and allows clock signal levels to be compatible with the CMOS clock receivers.

There is no system support for halting clocks, or for single-stepping or n-stepping clocks. The scan tools do, however, allow halting clocks within each of the scannable VLSI chips.

Scan Circuitry

The processor board's scan circuitry interfaces to the service processor's four-line serial scan port and enables the user, via the service processor, to scan test each of the VLSI chips and transceiver groups selectively. The arbitration and address buffer chip can be scanned independently, whereas the address (4) and data (8) bus transceivers are chained. This scan feature is used as a fault analysis tool in manufacturing.

Printed Circuit Board and Mechanical

The processor board uses a 12-layer construction and has an approximate overall thickness of 0.075 inch. Among the 12 layers are six signal layers, three ground layers, and three voltage plane layers. Cyanate ester dielectric material is used

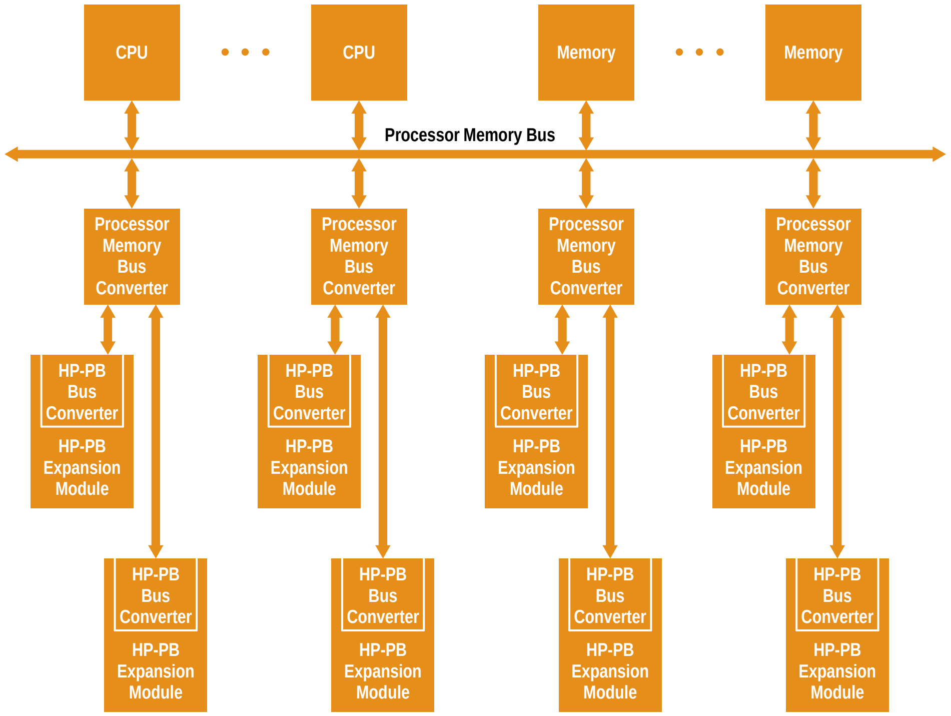


Fig. 12. Model T500 I/O subsystem.

for its faster signal propagation speed over FR-4 material and its ability to achieve reduced board thickness for a given trace impedance. The nominal signal trace impedance is 51 ohms for all high-speed signal nets.

Every attempt was made to keep high-speed signal traces closely coupled to a neighboring ground layer to minimize signal perturbations and EMI. Bypass capacitors are distributed liberally across the board to suppress high-frequency noise. EMI decoupling techniques consistent with the other Model T500 boards are used to direct common-mode noise to chassis ground.

The dimensions of the processor board are 16.90 inches by 7.35 inches. The two processor modules extend beyond the 7.35-inch dimension by approximately 3.25 inches and are supported by a sheet-metal extender which effectively makes the board assembly 14 inches deep. The modules are mounted parallel to the processor board and the sheet-metal extender and are secured by screws and standoffs. The sheet-metal extender also has a baffle which directs forced air across the modules for increased cooling.

Input/Output Subsystem

The HP 9000 Model T500 represents a major advance in the areas of high I/O throughput and highly scalable connectivity. The Model T500 system provides large aggregate I/O throughput through the replication of input/output buses with large slot counts. These I/O buses are arranged in a two-level tree. A bus converter subsystem connects the processor memory bus of the Model T500 system with the Hewlett-Packard precision bus (HP-PB) I/O buses, as shown in Fig. 12. The bus converter subsystem consists of a processor memory bus converter, a bus converter link (see Fig. 13), and an HP-PB bus converter. It translates the logical protocol and electrical signaling of data transfers between the processor memory bus and the I/O cards on the HP-PB bus.

The I/O subsystem guarantees data integrity and provides high reliability through parity protection of all data and transactions and through the hardware capability of online replaceable cards.

The bus converter subsystem is transparent to software under normal operating conditions. Each I/O module on an HP-PB bus in the system is assigned a range of physical memory addresses. I/O modules appear to software as sets of registers.

All modules can be DMA capable and generally implement scatter/gather DMA controllers. These scatter/gather DMA controllers allow virtually contiguous data located in physically noncontiguous pages to be transferred with minimal CPU assistance. A chain of DMA commands is written into memory by the processor. The I/O card is notified of the location of the chain and that it is ready for use. The I/O card then uses the scatter/gather DMA controller to follow the chain and execute the commands. In this manner the I/O card can write data (scatter) to different physical pages during the same DMA operation. The I/O card can also read data (gather) from different physical pages during the same DMA operation. When the I/O card finishes all of the commands in the chain, it notifies the processor, usually through an interrupt.

The processor memory bus converter is a dual bus converter that connects to two HP-PB buses through a pair of cables and the HP-PB bus converter. The HP-PB bus converter is plugged into a slot in an HP-PB expansion module and provides the central HP-PB bus resources of arbitration, clock generation, and online replacement signals in addition to the connection to the processor memory bus.

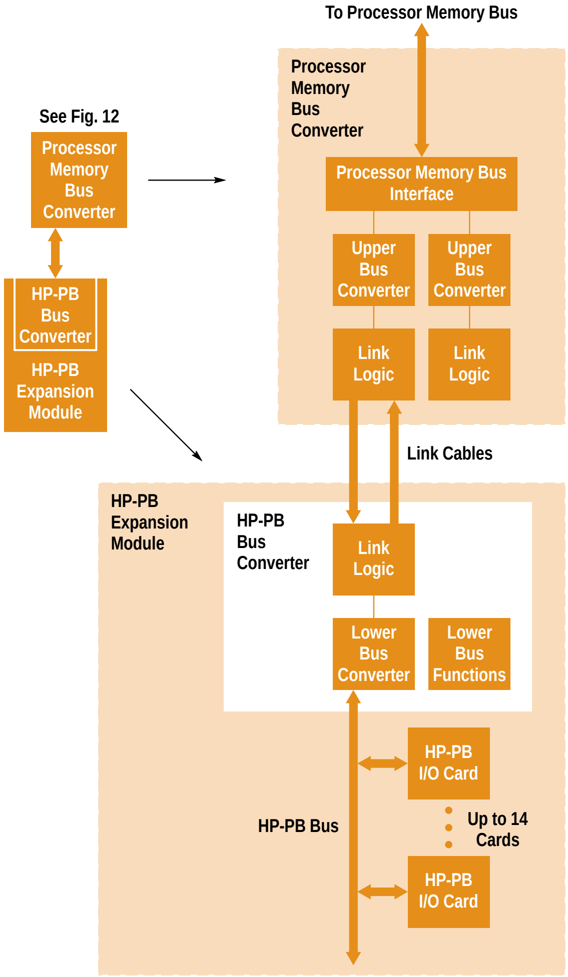


Fig. 13. Detail of Model T500 I/O subsystem.

Each HP-PB expansion module is a 19-inch rack-mountable assembly that connects any combination of up to 14 single-height or 7 double-height cards to the HP-PB bus. A Model T500 supports connection of 112 single-height HP-PB cards.

Each HP-PB bus is a 32-bit multiplexed address and data bus with byte-wise parity protection and additional parity protection across the control signals. The frequency of operation is fixed at 8 MHz, leading to a peak bandwidth of 32 Mbytes/s. The aggregate I/O rate for the Model T500 system is thus 256 Mbytes/s.

The HP-PB I/O function cards include SCSI, fast/wide SCSI, FDDI (doubly connected), Ethernet LAN, token ring LAN, HP-FL fiber-link disk connect, IEEE 488 (IEC 625), X.25 and other WAN connects, terminal multiplexer cards, and other I/O functions. Using HP-FL cards and HP C2250A disk arrays, the corporate business server hardware can support over 1.9 terabytes of disk storage on over 1000 disk spindles.

Processor Memory Bus Converter

The Model T500 accepts up to four processor memory bus converters plugged into the processor memory bus backplane. Each processor memory bus converter consists of two logically separate upper bus converter modules sharing a single bus interface (see Fig. 13). This reduces the electrical loading on the processor memory bus while providing the necessary fanout for a high-connectivity I/O subsystem.

The processor memory bus converter provides resource-driven arbitration and transaction steering on the processor memory bus for transactions involving the I/O subsystem. The processor memory bus converter provides a maximum bandwidth of 96 Mbytes/s. Transactions through the processor memory bus converter are parity protected, and error correcting code is generated and checked at the processor memory bus interface to guarantee data and transaction integrity.

The upper bus converter modules are implemented in custom CMOS26 VLSI chips in 408-pin pin-grid array packages. They arbitrate with each other for the processor memory bus interface chips on the processor memory bus side and implement the bus converter link protocol on the link side.

The processor memory bus interface consists of 12 bus transceiver chips (eight data and four address) and an arbitration and address buffer chip. These chips are used in a two-module mode. The data bus transceivers drive independent bidirectional data buses to the two upper bus converter module chips. The address bus transceivers drive a single unidirectional address to both bus converter chips, but receive independent address buses from the two upper bus converter chips.

The processor memory bus converter also provides discrete industry-standard logic to translate the bus converter link signals between the CMOS levels of the upper bus converter chip and the +5V ECL levels of the link cable.

Bus Converter Link

Each of the two upper bus converter modules connects through two cables to a lower bus converter module, the HP-PB bus converter (see Fig. 13). Each cable is a high-performance 80-conductor flat ribbon insulation displacement connector cable which allows the lower bus converter module and the HP-PB expansion module to be located up to 10 meters away. These cables and the protocol that is used on them make up the bus converter link.

The bus converter link protocol is a proprietary protocol allowing pipelining of two transactions with positive acknowledgment. The signals are point-to-point +5V ECL differential signals, two bytes wide and parity protected. The status information from the opposite bus converter module is embedded in the link protocol. The signaling rate across the bus converter link is one-half the processor memory bus frequency or 30 MHz in the Model T500 system. The peak bus converter link bandwidth is therefore 60 Mbytes/s with an average protocol overhead of 10%. The address overhead is on the order of 20% leaving an average data transfer rate of 42 Mbytes/s.

HP-PB Bus Converter

The HP-PB bus converter connects the bus converter link to the HP-PB bus in the HP-PB expansion module. In addition to the bus converter functions, the HP-PB bus converter provides the central resources for the HP-PB bus to which it connects, including bus clock generation, arbitration logic and online replacement power-on signals. The bus clock generation and arbitration are performed by discrete industry-standard components on the board. The HP-PB bus converter functions are implemented in a custom CMOS26 chip in a

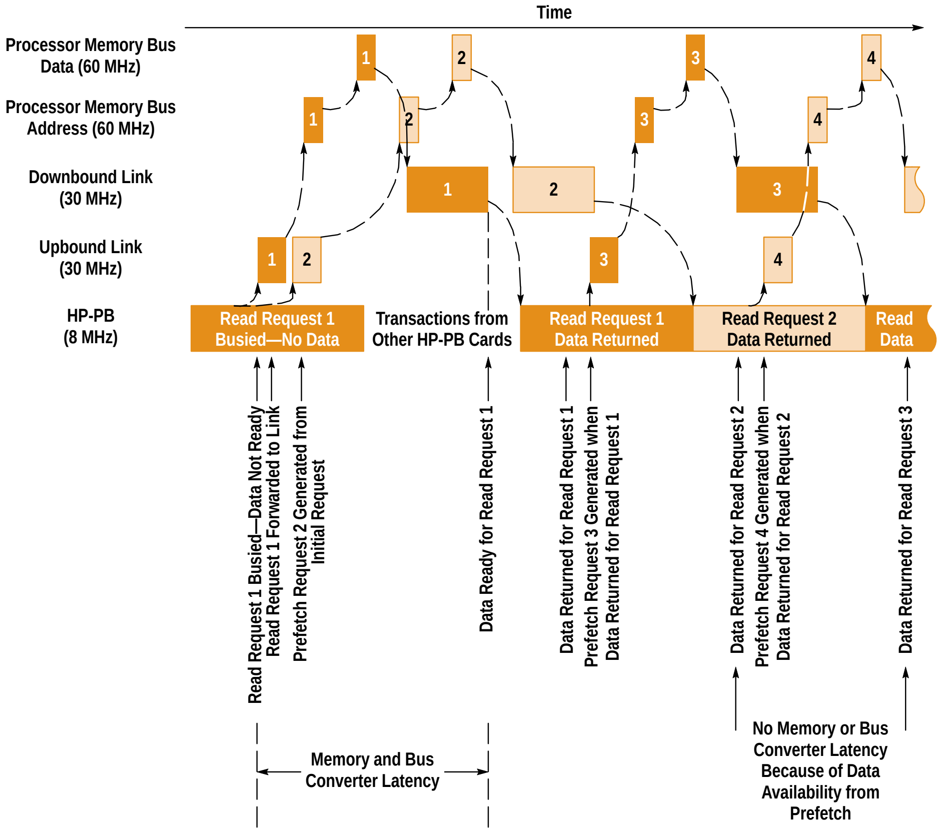

Time
Processor Memory Bus Data (60 MHz)
Processor Memory Bus Address (60 MHz)
Downbound Link (30 MHz)
Upbound Link (30 MHz)
HP-PB (8 MHz)
1
2
3
4
Read Request 1 Busied—No Data
Transactions from Other HP-PB Cards
Read Request 1 Data Returned
Read Request 2 Data Returned
Read Data
Read Request 1 Busied—Data Not Ready
Read Request 1 Forwarded to Link
Prefetch Request 2 Generated from Initial Request
Data Ready for Read Request 1
Data Returned for Read Request 1
Prefetch Request 3 Generated when Data Returned for Read Request 1
Data Returned for Read Request 2
Prefetch Request 4 Generated when Data Returned for Read Request 2
Data Returned for Read Request 3
Memory and Bus Converter Latency
No Memory or Bus Converter Latency Because of Data Availability from Prefetch

Fig. 14. Speculative prefetch.

272-pin pin-grid array package. Electrical signal level translation between the CMOS of the lower bus converter chip and the +5V ECL of the link cable is performed using the same discrete industry-standard components as are used on the processor memory bus converter. The HP-PB bus converter acts as a concentrator for the I/O traffic from the HP-PB cards bound for the system memory or the processors.

The HP-PB bus converter implements a speculative prefetch for DMA reads of memory by HP-PB cards (data transferred from memory to an I/O device under the I/O card's control). This provides greater performance by offsetting the transaction and memory latency. The prefetch algorithm always has two read requests in the transaction pipeline to memory (see Fig. 14). When a read transaction to memory is accepted for forwarding by the HP-PB bus converter, it forwards the first read and then issues a second read request with the address incremented by the length of the original read transaction. As the data is returned to the requester, a new read transaction with the address incremented by twice the length of the transaction is issued on the bus converter link. The prefetching stops when the I/O card does not request the next read in the next transaction interval on the HP-PB bus or when the address generated would cross a 4K page boundary. Speculative prefetch increases the possible read data bandwidth from 3 Mbytes/s to over 18 Mbytes/s.

The HP-PB bus converter supports DMA writes at the full HP-PB data bandwidth of 18 Mbytes/s for 16-byte writes and 23 Mbytes/s for 32-byte writes. The difference between the peak bandwidth and the data bandwidth represents the effects of the address overhead and bus turnaround cycles.

The HP-PB bus converter carries parity through the entire data path and checks the parity before forwarding any transaction onto the link or the HP-PB bus to guarantee data and transaction integrity.

The HP-PB bus converter and HP-PB backplane in the HP-PB expansion module together provide the hardware and mechanisms to allow online replacement of HP-PB I/O cards. The HP-PB bus converter provides a read/write register through which the power-on signal to each HP-PB card can be controlled independently. When this signal is deasserted to an HP-PB card, the card's bus drivers are tristated (set to a high-impedance state) and the card is prepared for withdrawal from the HP-PB expansion module. The HP-PB backplane provides the proper inductance and capacitance for each slot so that a card can be withdrawn while the system is powered up without disturbing the power to the adjacent cards. The hardware online replacement capability makes possible future enhancements to the Model T500 for even higher availability.

Logic in the HP-PB expansion module monitors the ac power into the module and indicates to the HP-PB bus converter via a backplane signal when power is about to fail or when the dc voltages are going out of specification. The powerfail warning signal is passed up through the bus converter modules to allow the Model T500 system to prevent corruption of the machine state.

HP Precision Bus

The HP-PB is a multiplexed 32-bit address and data bus with a fixed clock rate of 8 MHz. The HP-PBs in the Model T500 system are completely independent of the processor memory bus clocks. The HP-PB bus converter synchronizes the data between the HP-PB and the bus converter link.

The HP-PB provides for global 32-bit addressing of the I/O cards and for flexibility in address assignment. Each HP-PB is allocated a minimum of 256K bytes during configuration. This address space is evenly divided between 16 possible slots. Each slot on the HP-PB supports up to four I/O modules, each of which is allocated a 4K-byte address space. This 4K-byte space is called the hard physical address space. Any module that requires additional address space is assigned address space at the next available bus address. This additional address space is called the soft physical address space. Soft physical address space assigned to all I/O modules on a single HP-PB is contiguous. The processor memory bus converter determines if a transaction is bound for a given HP-PB by checking for inclusion in the range determined by the hard physical address and soft physical address space of the HP-PB.

The hard physical address of an I/O card contains the control and status registers defined by the PA-RISC architecture through which software can access the I/O card. Each HP-PB card has a boot ROM called the I/O dependent code ROM, which is accessed by indirection through a hard physical address. This ROM contains the card identification, configuration parameters, test code, and possibly boot code. The I/O dependent code ROM allows I/O cards to be configured into a system before the operating system is running and allows the operating system to link to the correct driver for each card.

The HP-PB transaction set is sufficiently rich to support efficient I/O. There are three classes of transactions: write, read, and clear or semaphore. Each transaction is atomic but the HP-PB bus protocol provides buffered writes for high performance and provides a busy-retry capability to allow reads of memory to be split, providing parallelism and higher bandwidth. Each HP-PB transaction specifies the data payload. The transaction set supports transactions of 1, 2, 4, 16, and 32 bytes. DMA is performed using 16-byte or 32-byte transactions initiated under the control of the I/O card. Each HP-PB transaction contains information about the master of the transaction so that errors can be reported and data easily returned for reads.

The HP-PB and I/O subsystem provides an efficient, flexible, and reliable means to achieve high I/O throughput and highly scalable connectivity.

Memory System

The memory subsystem for the HP 9000 Model T500 corporate business server uses 4M-bit DRAMs for a 256M-byte capacity on each board. It is expandable up to 2G bytes of error-correcting memory. To minimize access latency in a multiprocessor environment, the memory subsystem is highly interleaved to support concurrent access from multiple processors and I/O modules. A single memory board can contain 1, 2, or 4 interleaved banks of 64M bytes. The combination of interleaving and low latency for the board provide a bandwidth of 960 Mbytes/s. Furthermore, different-sized memory boards using different generations of DRAMs can coexist in the system, allowing future memory expansion while preserving customer memory investments.

From the standpoint of complexity, the memory board is the most sophisticated board in the Model T500 system. To meet its performance requirements, the design uses leading-edge printed circuit technologies and new board materials. These are described under “Manufacturing” later in this article. The memory board includes 4273 nets (or signals), 2183 components, and over 28,850 solder joints. Double-sided surface mount assembly provides high component density. The 2183 components are mounted in an area of only 235 square inches.

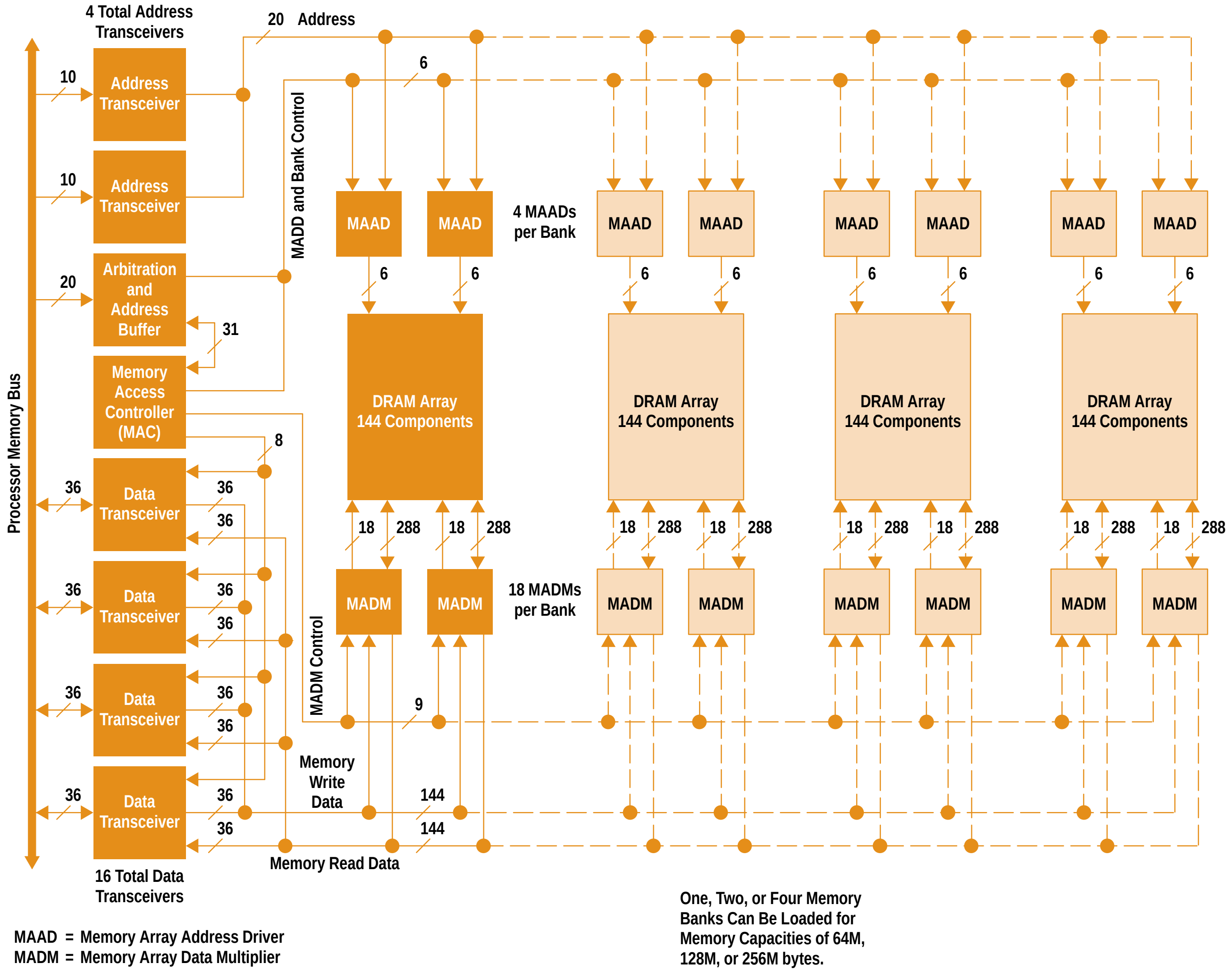
The processor memory bus electrical design limits the length of the bus for 60-MHz operation to 13 inches. Consequently, the memory board design is considerably constrained. The limited number of slots requires the capacity of each memory board to be high. The short bus length makes each of the slots narrow, forcing a low profile for each memory board. Bus transceivers are located close to the connector on each daughter card to keep stub lengths to a minimum.

Memory Interleaving

Memory boards are manufactured in 64M-byte, 128M-byte, and 256M-byte capacities. The 64M-byte and 128M-byte memory capacities are achieved by partially loading the 256M-byte board. Memory interleaving tends to distribute memory references evenly among all blocks in the system. In the event that two processors desire to access memory in consecutive quads, interleaving provides that the second access will likely be to an idle bank. The memory design for the Model T500 allows the benefits of interleaving to be based on the total number of memory banks installed in the system, regardless of the number of boards that the banks are spread across.[9] The processor memory bus protocol maximizes performance by interleaving all the banks evenly across the entire physical address space, regardless of the number of banks. This is superior to interleaving schemes that limit the effect of interleaving to numbers of banks that are powers of two.

Memory Board Partitioning

Partitioning of the memory board into VLSI chips follows the requirements of the DRAMs and the bank organization. This partitioning is illustrated in the memory board block diagram, Fig. 15. 256M-byte capacity with single-bit error correction requires 576 4M-bit DRAMs, each of which is organized as 1M by 4 bits. 64-byte data transfers and minimized latency require a 576-bit bidirectional data bus for each bank’s DRAMs. The effort to minimize latency and the restriction of the processor memory bus to narrow slots prevented the use of SIMM modules similar to those used in PCs and workstations. The fixed timing relationships on the processor memory bus required that there be four of these 576-bit data buses for the four banks on the 256M-byte memory board to prevent contention between writes to one bank and reads from another bank. A multiplexing function is provided



MAAD = Memory Array Address Driver
MADM = Memory Array Data Multiplier

One, Two, or Four Memory Banks Can Be Loaded for Memory Capacities of 64M, 128M, or 256M bytes.

Fig. 15. Memory board block diagram.

between the four slow 576-bit DRAM data buses and the 60-MHz 128-bit data bus of the processor memory bus.

To implement these requirements, a set of five VLSI chips is used. As identified on the block diagram, these are:

- Bus transceivers. This design is also used on the processor and bus converter boards.
- Arbitration and address buffer. This chip provides for arbitration and acts as an additional pair of address transceivers. This design is also used on the processor and bus converter boards.
- Memory array data multiplexer (MADM). This chip multiplexes the slow DRAM data signals to a pair of unidirectional 60-MHz, 128-bit buses to and from the data transceivers.
- Memory array address driver (MAAD). This chip drives address and RAS and CAS to the DRAMs. It is a modified version of a standard commercial part.
- Memory access controller (MAC). This chip provides the overall control function for the memory board. In particular, the MAC implements the required architectural features of the memory system and controls DRAM refresh.

Except for the MAAD, which is in a 44-pin PLCC (plastic leaded chip carrier), each of these ICs is a fine-pitch, quad flatpack (QFP) component, with leads spaced 0.025 inch apart. The bus transceiver and MADM are packaged in 100-pin QFPs and the arbitration and address buffer and MAC are in 160-pin QFPs. The full 256M-byte board includes 20 bus transceivers, one arbitration and address buffer, 72 MADMs, 16 MAADs, and one MAC as well as the 576 4M-bit DRAM chips.

Fig. 16 is a photograph of the 256M-byte memory board.

Printed Circuit Board Design

In addition to restrictions on the memory board caused by the processor memory bus design, there were a significant number of other electrical design and manufacturing requirements on the board. The onboard version of the processor memory bus address bus is a 31.70-inch, 60-MHz unidirectional bus with 16 loads on each line. There are two 128-bit, 60-MHz, 9.15-inch buses with five loads on each line. With the large number of components already required for the board, it would not have been feasible to terminate these buses. The clock tree for the VLSI on the board feeds a total of 94 bidirectional shifted ECL-level inputs and 16 single-ended inputs, with a goal of less than 250 ps of skew across all 110 inputs. The size chosen for the memory board is 14.00 by 16.90 inches, the maximum size allowed by surface mount equipment for efficient volume production. Restriction

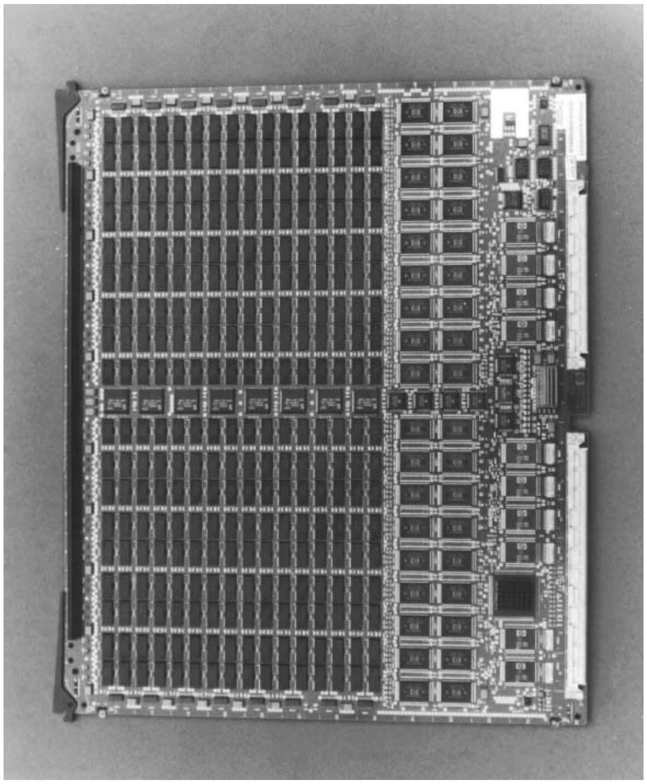


Fig. 16. 256M-byte memory board.

to this size was an important factor in almost every design decision made for the board.

Preliminary designs of critical areas of the board showed that the densest feasible routing would be required. Leading-edge HP printed circuit production technology allows a minimum of 0.005-inch lines and 0.005-inch spaces. Vias can be either 0.008-inch finished hole size with 0.021-inch pads, or 0.012-inch finished hole size with 0.025-inch pads. Both of these alternatives are currently limited to a maximum aspect ratio of 10:1 (board thickness divided by finished hole size). The aspect ratio also influences the production cost of the board significantly because of plating yields, as well as the achievable drill stack height.

With the given layout conditions, several trade-off studies were done to find the best alternative in terms of electrical performance, manufacturing cost for the loaded assembly, reliability, and risk for procurement and process availability at both fabrication and assembly. The best alternative finally uses the leading-edge layout geometries, eight full signal layers, and two partial signal layers. Since the initial projections of the number of layers required to route the board led to an anticipated board thickness greater than 0.080 inch, the aspect ratio requirements caused the 0.008-inch finished hole size via option to be rejected. Even with 0.025-inch pads and 0.012-inch finished hole size vias, the aspect ratio approaches 10. Therefore, a sophisticated board material is required to prevent thermal cycling from stressing vias and generating distortions on the board by expansion of the thickness of the board. Cyanate ester material (HT-2) was chosen over other substrate alternatives because of its superior electrical and mechanical performance.[10]

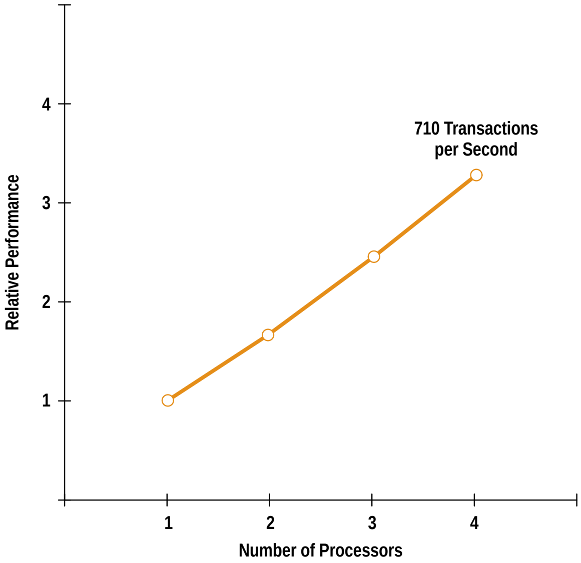


Fig. 17. Scaling of online transaction processing (OLTP) performance with number of processors.

Multiprocessor Performance

Performance

An HP 9000 Model T500 corporate business server, a six-processor, 90-MHz PA-RISC 7100 CPU with a 60-MHz bus, achieved 2110.5 transactions per minute (U.S.$2,115 per tpmC) on the TPC-C benchmark.[5] In the following discussions, the available multiprocessor performance data is a mixture of data from both the Model T500 and the older Model 890 systems.

Data for the HP 9000 Model 890 (the precursor of the Model T500, which uses one to four 60-MHz PA-RISC processors and the same memory, bus, and I/O subsystems as the Model T500) is available for the TPC-A benchmark and one to four processors. Fig. 17 shows how multiprocessing performance scales on a benchmark indicative of OLTP performance.[6]

The SPECrate performance for the Model T500 is shown in Fig. 18.[7] The SPEC results show linear scaling with the number of processors, which is expected for CPU-intensive workloads with no mutual data dependencies. The OLTP benchmarks are more typical for real commercial applications. The

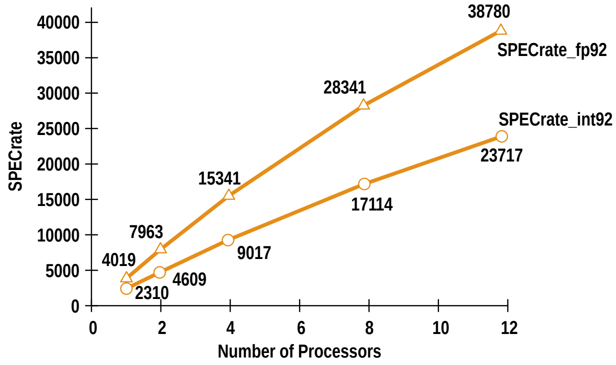


Fig. 18. Model T500 SPECrate performance.

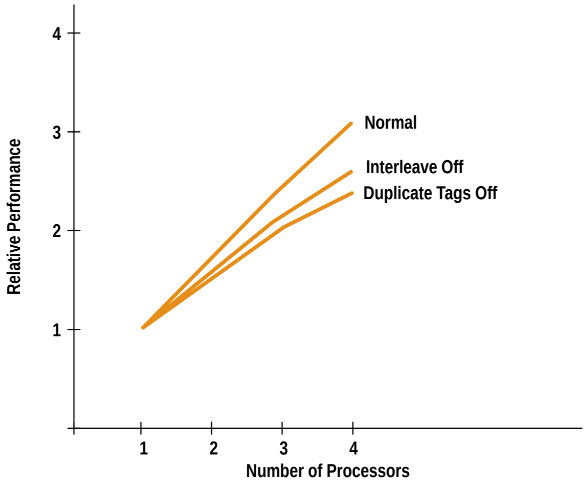


Fig. 19. Model 890 program development performance.

losses in efficiency are caused by factors such as serialization in the I/O subsystem and contention for operating system resources.

Fig. 19 shows the performance of the Model 890 on an HP-internal benchmark representative of 24 interactive users executing tasks typical of a program development environment.

The benchmark results confirm the value of key design decisions. For example, nearly all transactions were useful—only 6% of all transactions were busied and only 1.5% of all bus quads were waited. Disabling the interleaving or disabling the duplicate cache tags did not affect bus utilization.

The efficiency of the bus was reflected in system throughput. Normal operation showed near-linear multiprocessor scaling through four processors. Changing the interleaving algorithm from the normal case of four blocks interleaved four ways to four blocks not interleaved caused a significant performance impact. As expected, the penalty was greater at higher degrees of multiprocessing, peaking at a penalty of 15% in a four-processor system. Disabling the duplicate cache tags incurred an even greater cost: the decrease in system performance was as much as 22%, with the four-processor system again being the worst case.

These tests showed that the high-speed pipelined processor memory bus, fast CPUs with large caches, duplicate cache tags in the processor interfaces, and highly interleaved large physical memory allow the Model T500 system to scale efficiently up to twelve-way multiprocessing.

Service Processor

As part of the challenge of producing the HP 9000 Model T500 corporate business server, targeted at demanding business applications, it was decided to try to make a significant improvement in system hardware availability. Hardware availability has two components: mean time between failures (MTBF), which measures how often the computer hardware fails, and mean time to repair (MTTR), which measures how long it takes to repair a hardware failure once one has occurred. The service processor makes a significant improvement in the MTTR portion of the availability equation by reducing the time required to repair the system when hardware failures do occur.

HP's computer systems are typically supported from our response centers, where HP has concentrated some of the most knowledgeable and experienced support staff. These support engineers generally provide the first response to a customer problem. They make the initial problem diagnosis and determine which of HP's resources will be applied to fixing the customer's system. The greatest opportunity to improve the system's MTTR existed in improving the ability of the support engineers at the response centers to access failure information and control the system hardware. The following specific goals were set:

- All of the troubleshooting information that is available locally (at the failed system) should be available remotely (at the response center).
- Information should be collected about hardware failures that prevent the normal operating system code from starting or running.
- Information about power and environmental anomalies should be collected.
- Information about operating system state changes should be collected.
- Error information should be available to error analysis software running under the operating system if the operating system is able to recover after an anomaly occurs.
- A means should exist to allow support personnel to determine the system hardware configuration and alter it without being present at the site to allow problems to be worked around and to aid in problem determination.
- The support hardware should be as independent of the remainder of the computer system as possible, so that failures in the main hardware will not cause support access to become unavailable.
- Error reporting paths should be designed to maximize the probability that failure symptoms will be observable even in the presence of hardware failures.
- Failure in the support hardware should not cause failure of the main computer system.
- Failure of the support hardware should not go unnoticed until a failure of the main system occurs.
- The hardware support functions should be easily upgradable without requiring a visit by support personnel and without replacing hardware.

Hardware Implementation

The above goals are achieved by providing a single-board service processor for the Model T500 system. The service processor is a microprocessor-controlled board that is located in the main cardcage. This board has control and observation connections into all of the hardware in the main cardcage. This board also contains the power system control and monitor which controls the power system. The service processor has a command-oriented user interface which is accessible through the same console mechanism as the operating system console connections on previous systems (through the system's access port). The logical location of the service processor is shown in Fig. 20.

The service processor and power system control and monitor are powered by special bias power which is available whenever ac power is applied to the system cabinet. The service

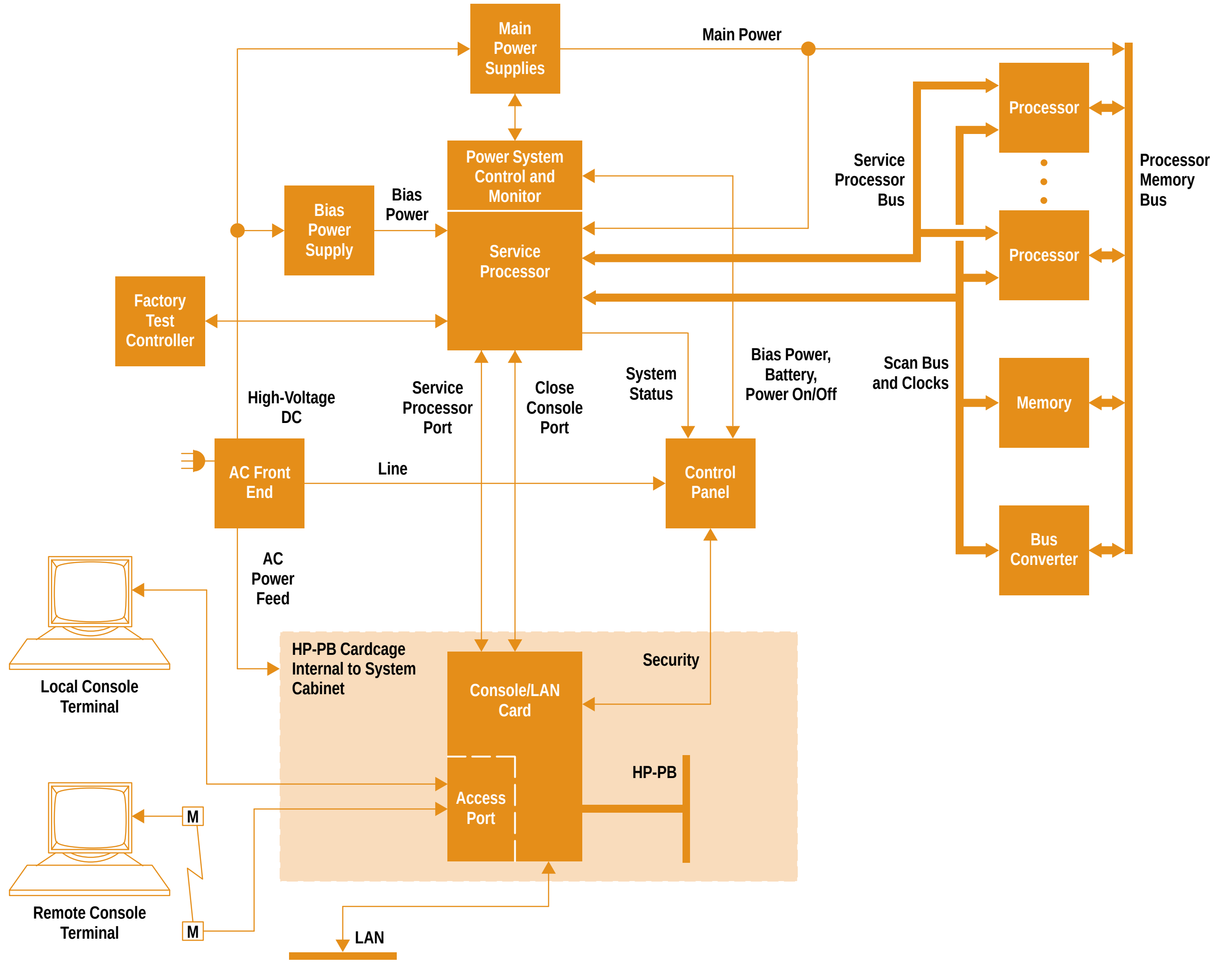


Fig. 20. Service processor block diagram.

processor is thus independent of the main system power supplies, and can be accessed under almost all system fault conditions.

The service processor has a communications channel to the power system control and monitor which allows it to provide the operating code for the power system control and monitor microprocessor, and then to issue commands to the power system control and monitor and monitor its progress. The power system control and monitor controls the power system under service processor supervision and notifies the service processor of power and environmental problems. The service processor provides a user interface to the power system which is used by support personnel when troubleshooting power and environmental problems.

The service processor is connected to each card on the processor memory bus by both the system clocks and the scan bus. Through the system clocks, the service processor provides clocking for the entire Model T500 system. The scan bus allows the service processor to set and read the state of the cards without using the main system bus. This mechanism is used to determine and alter system configuration and for factory testing.

The service processor is connected to the processors in the system by the service processor bus. The service processor bus allows the processors to access instructions and data stored on the service processor. The instructions include processor self-test code and processor dependent code, which performs architected system functions. The data stored on the service processor includes configuration information and logs of system activity and problems. The service processor bus also allows the processors to access common system hardware that is part of the service processor, such as system stable storage which is required by the PA-RISC architecture, and provides access to the console terminals through the close console port. Because service processor bus access is independent of the condition of the processor memory bus, the processors can access error handling code, make error logs, and communicate with the console terminals even if the processor memory bus has totally failed.

The service processor drives the system status displays on the control panel. These include the large status lights, the number of processors display, and the activity display. The service processor also mirrors this information onto the console terminals on the status line.

The connections between the service processor and the console/LAN card provide several functions. The service processor's user interface is made available on the local and remote console terminals by the access port firmware which is part of the console/LAN card. The user interface data is carried through the service processor port connection. Because the internal HP-PB cardcage which houses the console/LAN card is powered by the same source of ac power as the service processor, the access port and its path to the console terminals are functional whenever the service processor is powered. The system processors access the console terminals through the close console port connection to the access port firmware during the early stages of the boot process and during machine check processing when the I/O subsystem is not necessarily functional. The service processor also sends control information and communicates its status to the console/LAN card through the service processor port. Console terminal access to the system and service processor functions is controlled by the access port firmware on the console/LAN card.

A connection exists between the service processor and a test controller used for system testing in the factory. This connection allows the system internal state to be controlled and observed during testing.

Because the service processor and power system control and monitor do not operate from the same power supplies as the processor memory bus, the service processor's control features and error logs are available even when the remainder of the system is inoperable. Because logging, error handling, and console communications paths exist that are independent of the system buses, these functions can operate even when system buses are unusable. The service processor is architected so that its failure does not cause the operating system or power system to fail, so that failure of the service processor does not cause the system to stop. The access port is independent of the service processor and detects service processor failure. It notifies the user of service processor failure on the console terminals, providing time for the service processor to be repaired before it is needed for system-critical functions.

Features

The hardware implementation described above is extremely flexible because of its large connectivity into all of the main system areas. As a result, the service processor's features can be tailored and changed to ensure that the customer's service needs are adequately met. The service processor in its current implementation includes the service features described in the following paragraphs.

Configuration Control. The service processor keeps a record of the processor memory bus configuration including slot number, board type, revision, and serial number. The service processor reconciles and updates this information each time the system is booted by scanning the processor memory bus and identifying the modules it finds. Various error conditions cause defective processor memory bus modules to be automatically removed from the configuration. The user is alerted to such changes and boot can be optionally paused on configuration changes. The service processor's user interface contains commands to display and alter the configuration, including removing modules from the configuration or adding them back into the configuration. Modules that are removed no longer electrically affect the system, making configuration an effective means of remotely troubleshooting problems on the processor memory bus.

Logs. The service processor has a large log area that contains logs of all service-processor-visible events of support significance. Each log contains the times of event occurrences. Logs that warn of critical problems cause control panel and console terminal indications until they have been read by the system operator. The service processor user interface contains commands to read and manage the service processor logs. Information in the service processor logs can be accessed by diagnostic software running under the operating system. The service processor logs include:

- Power system anomalies
- Environmental anomalies
- Ac power failure information
- Automatic processor memory bus module deconfigurations that occur because of failures
- Operating system major state changes (such as boot, testing, initialization, running, warning, shutdown)
- High-priority machine check information
- Problems that occur during system startup before the processors begin execution
- Processor self-test failure information.

Operating System Watchdog. The service processor can be configured to observe operating system activity and to make log entries and control panel and console indications in the event of apparent operating system failure.

Electronic Firmware Updates. The service processor and processor dependent code work together to update system firmware without the need for hardware replacement. The service processor contains the system processor dependent code (boot and error firmware), the firmware for the power system control and monitor to control the power system, and its own firmware. Two copies of each exist in electrically erasable storage so that one copy can be updated while the other copy is unchanged. The service processor can switch between the two copies in case a problem occurs in one copy.

Remote Access. The user gains access to the service processor user interface through the access port. The access port is the single point of connection for the system console terminals, both local and remote. As a result, all troubleshooting information that is available on local console terminals is available remotely.

Factory Test Support. The service processor serves as a scan controller, providing full access to the internal state of the custom VLSI chips contained on processor memory bus cards. This access is provided through the programmable clock system and the scan bus. Using the scan controller features of the service processor, a factory test controller can test the logic in the processor memory bus portion of the system under automatic control.

System Status Control. Because the service processor controls the system status indicators, it is able to display an accurate summary of the complete hardware and software state of the

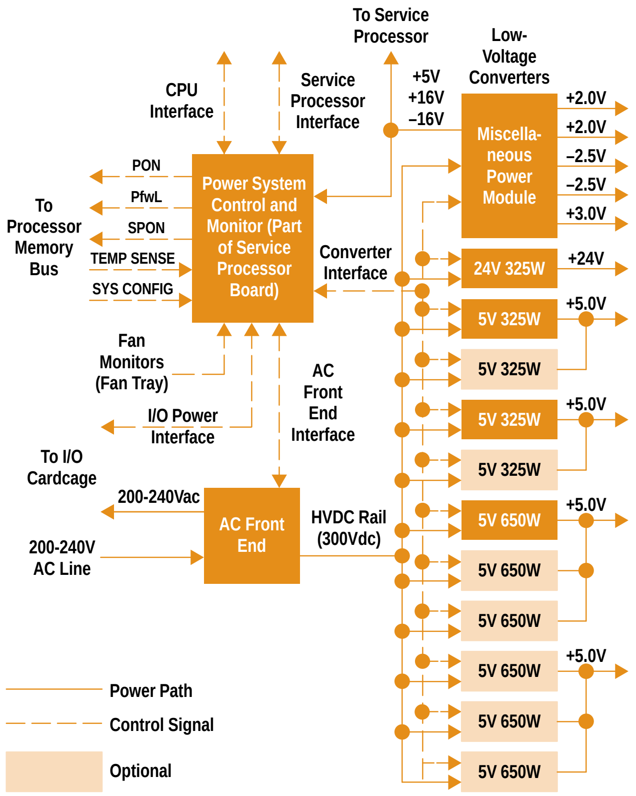


Fig. 21. Power system block diagram.

system. The service processor can do this even when the system processors or main power system are unable to operate.

Power System

The power system provides regulated low-voltage dc to all logic assemblies in the processor memory bus cardcage, and to the array of fans located just below the cardcage assembly. The power system is designed to grow and reliably support the need for ever-increasing processor, memory, and I/O performance. It has the capacity to deliver almost 4,000 watts of dc load power continuously. The block diagram of the power system is shown in Fig. 21. The modular design consists of an ac front-end assembly, several low-voltage dc-to-dc converters, and a power system control and monitor built within the service processor.

The ac front end, shown in Fig. 22, contains one to three power-factor-correcting upconverter modules, each providing regulated 300Vdc, and has an output capacity of 2.2 kilowatts. The upconverter modules run on single-phase or dual-phase, 208Vac input. They have output ORing diodes, implement output current sharing, and are capable of providing true N+1 redundancy for higher system capacity and availability. (N+1 redundancy means that a system is configured with one more module than is necessary for normal operation. If there is a subsequent single module failure the extra module will take over the failed module's operation.)

The active power-factor-correcting design allows the product to draw near unity power factor, eliminating the harmonic currents typically generated by the switching power supplies in a computer system. The design also has a very wide ac input operating range, is relatively insensitive to line voltage transients and variations, and allows a common design to be used worldwide. It also provides a well-regulated 300Vdc output to the low-voltage dc-to-dc converters.

The low-voltage dc-to-dc converters are fed from a single 300V rail and deliver regulated dc voltage throughout the main processor cardcage. The single-output converters, of which there are two types, have capacities of 325 and 650 watts and a power density of about 3 watts per cubic inch. They have current sharing capability for increased output capacity, and are designed to recover quickly in the event of a module failure in a redundant configuration. The converters have output on/off control and a low-power mode to minimize power drain on the 300V rail when shut down. Their output voltage can be adjusted by the power system control and monitor.

The power system control and monitor provides control for power sequencing, fan speed control, and temperature measurement. It ensures that the modular converters and the system load are consistent with each other. The controller also monitors status and system voltages. This information is communicated to the service processor and saved in a log to aid in the support and maintenance of the system.

Together, the power system control and monitor, power-factor-correcting upconverters, and low-voltage dc-to-dc converters form a scalable, high-capacity, highly available, modular power system. The system is easily updated and can be upgraded to support higher-performance processor,

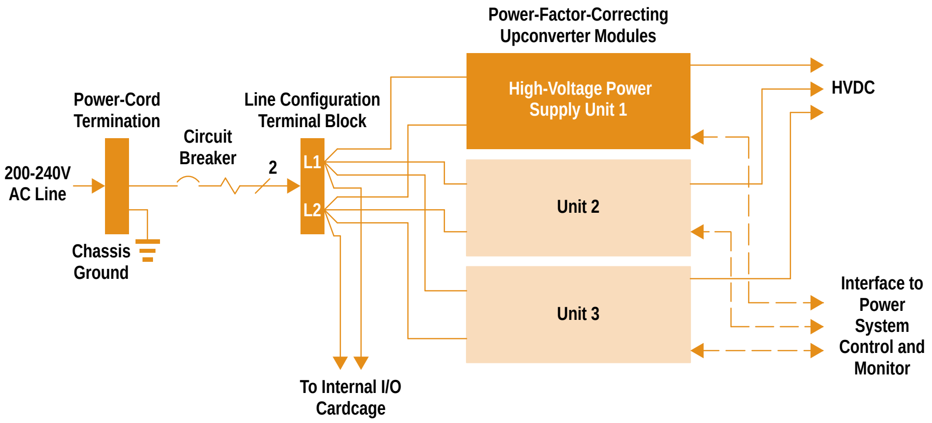

Power-Factor-Correcting Upconverter Modules
High-Voltage Power Supply Unit 1
Unit 2
Unit 3
HVDC
Power-Cord Termination
Line Configuration Terminal Block
Circuit Breaker
2
L1
L2
200-240V AC Line
Chassis Ground
To Internal I/O Cardcage
Interface to Power System Control and Monitor

Fig. 22. Ac front end block diagram.

memory, and I/O technologies as they are developed for the Model T500 platform.

Product Design

The Model T500 package is a single-bay cabinet. Overall, it is 750 mm wide by 905 mm deep by 1620 mm tall. A fully loaded cabinet can weigh as much as 360 kg (800 lb). A skeletal frame provides the cabinet's structure, supporting the card cages, fan tray, and ac front end rack. External enclosures with vents and a control panel attach to the frame.

The processor memory bus boards and the low-voltage dc-to-dc converters reside in cardcages in the upper half of the Model T500 cabinet. They plug into both sides of a vertically oriented, centered backplane to meet bus length restrictions. A bus bar assembly attaches to the upper half of the backplane to distribute power from the larger 650-watt converters to the extended-power slots that the processor boards use.

There are 16 processor memory bus slots in the Model T500: six in the front cardcage and ten in the rear cardcage. Eight of the 16 slots are extended-power slots, which have a board-to-board pitch of 2.4 inches, twice the 1.2-inch pitch of the other eight standard slots. These wider slots allow increased cooling capability for the processor board heat sinks. The standard slots are used for bus converters. Memory boards can go in either standard slots or extended-power slots.

Looking at the front view of the cabinet in Fig. 23, six extended-power processor memory bus slots are to the left of the low-voltage dc-to-dc converter cardcages in which four 650-watt converters reside above two 325-watt converters and the miscellaneous power module. When viewing the rear of the cabinet in Fig. 24, ten processor memory bus slots, two of which are extended-power, reside to the right of the converter cardcages in which four 650-watt converters are above three 325-watt converters. The service processor is located in a dedicated slot between the rear processor memory bus and the converter cardcages.

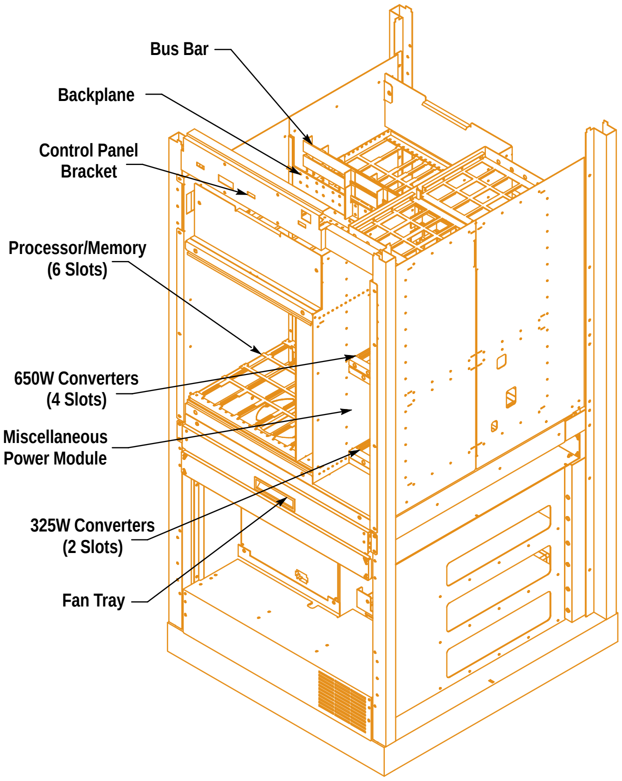

Bus Bar
Backplane
Control Panel Bracket
Processor/Memory (6 Slots)
650W Converters (4 Slots)
Miscellaneous Power Module
325W Converters (2 Slots)
Fan Tray

Fig. 23. Front view of Model T500 cabinet.

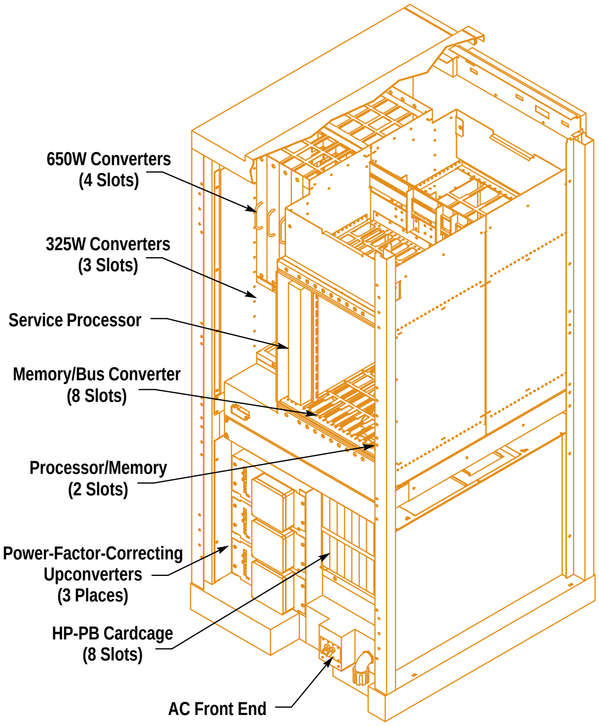

650W Converters (4 Slots)
325W Converters (3 Slots)
Service Processor
Memory/Bus Converter (8 Slots)
Processor/Memory (2 Slots)
Power-Factor-Correcting Upconverters (3 Places)
HP-PB Cardcage (8 Slots)
AC Front End

Fig. 24. Rear view of Model T500 cabinet.

The fan tray is located beneath the cardcages. Air enters through the top vents of the cabinet and is pulled through air filters and then through the processor memory bus and dc-to-dc converter cardcages to the fan tray. Half of the air is exhausted through the lower cabinet vents while the other half is directed to cool the HP-PB cardcage boards located in the ac front end rack. The fan tray is mounted on chassis slides to allow quick access to the fans.

The ac front end rack is mounted on the base of the Model T500 cabinet. This rack holds up to three power-factor-correcting power supply modules, an internal HP-PB cardcage, and the ac input unit. The HP-PB power supply has its own integral cooling fan. The ac front end power-factor-correcting modules have their own fans and air filters and take in cool air from the rear lower portion of the cabinet and exhaust air out at the front lower portion of the cabinet.

The rear of the internal HP-PB cardcage has an HP-PB bus converter and seven double-high or 14 single-high HP-PB slots as well as the battery for battery backup. The front of the HP-PB cardcage has a power supply and the power system control and monitor module. HP-PB backplane insertion is from the top of the cardcage by way of a sheet-metal carrier.

Additional rackmount HP-PB expansion modules and system peripherals are housed in peripheral racks. Both HP-PB card cages (internal and rackmount) leverage the same power supply and backplane assemblies, but have different overall package designs. The rackmount version has a cooling fan that directs air in a front-to-back direction. The HP-PB boards mount in a horizontal orientation and the cables exit towards the rear of the peripheral rack. The rackmount unit is 305 mm high (7 EIA standard increments) by 425 mm wide by 470 mm deep. The peripheral racks are 600 mm wide by 905 mm deep by 1620 mm tall and have mountings to hold products conforming to the EIA 19-inch standard.

The Model T500 industrial design team drove the system packaging design to come up with a unified appearance for HP's high-end and midrange multiuser systems. The result is an industrial design standard for a peripheral rack system that fits well with the Model T500 design. This cooperative effort ensured consistency in appearance and functionality.

Electromagnetic Compatibility

EMC shielding takes place at the printed circuit board level, the power supply level, and the cardcage level and does not rely on external enclosures for containment. This keeps noise contained close to the source. A hexagonally perforated metal screen is used above and below the processor memory bus cardcage to minimize resistance to airflow while providing the required EMI protection. Nickel plating is used on the steel cardcage pieces to ensure low electrical resistance between mating metal parts. The backplane has plated pads in the areas that contact the cardcage pieces. Conductive gaskets are used to ensure good contact between the backplane, the cardcages, and the cover plates. ESD (electrostatic discharge) grounding wrist straps are provided in both the front and rear of the cabinet.

Surface mount filtering is used on the backplane to control noise on signal lines exiting the high-frequency processor memory bus cardcages and to prevent noise from coupling into the low-voltage dc-to-dc converters.

All processor memory bus boards have a routed detail in four corner locations along the board perimeter to allow for grounding contact. A small custom spring fits into via holes and resides in the routed-out space. This spring protrudes past the edge of the board and contacts the card guides in the cardcage. Surface mount resistance elements lie between the vias and the board ground planes. This method of grounding the processor memory bus boards helps reduce EMI emissions.

System Printed Circuit Boards

The processor memory bus cardcage is designed to accept 16.9-inch-high-by-14-inch-deep boards. This large size was required for the 256M-byte memory board. Since the processor and processor memory bus converter did not require large boards, it was important to have a cardcage and cover plate design that allows boards of various depths to be plugged into the same cardcage, thereby optimizing board panel use.

The processor plugs into this deep cardcage by means of a sheet-metal extender. The bus converter was more difficult to accommodate since this shallow board requires cables to attach to its frontplane. Therefore, a transition plate was developed to transition from the shallower board bulkheads to the full-depth cardcage cover plates as shown in Fig. 25. This transition plate locks into the adjacent bulkhead to maintain the EMI enclosure. However, either the transition plate or the adjacent bulkhead to which it latches can be removed without disturbing the other.

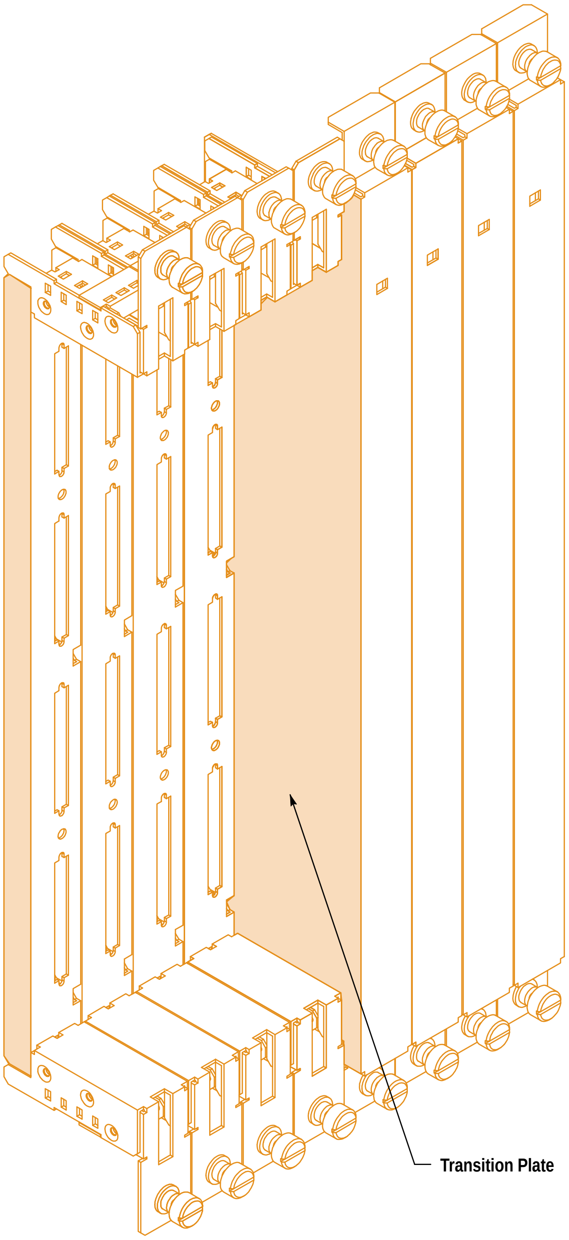


Fig. 25. Bus converter sheet-metal design, showing transition plate.

The Model T500 backplane is 25.3 inches wide by 21.7 inches high by 0.140 inch thick and has 14 layers. This backplane has many passive components on both sides, including press-fit and solder-tail connectors, surface mount resistors and capacitors, processor bus bars, and filters. The backplane connectors are designed to allow at least 200 insertions and withdrawals.

A controlled-impedance connector is used on the processor memory bus boards to mate to the backplane. The codevelopment that took place with the connector supplier was a major undertaking to ensure that the connector would work in our surface mount processes repeatably and meet our reliability and serviceability requirements.

Cooling

The Model T500 cooling system is designed to deliver high system availability. This is achieved by incorporating redundant fans, fan-speed tachometers, air temperature sensors on the hottest parts of the boards, and multiple-speed fans. The Model T500 meets the HP environmental Class C2 specification for altitudes up to 10,000 feet with an extension in temperature range up to 40°C.

Computational fluid dynamics software and thermal analysis spreadsheets were used to evaluate various components, heat sinks, and board placements. These tools helped the team make quick design decisions during the prototype stages. All high-powered components that were calculated to operate close to their maximum allowable junction temperature in the worst-case environment were packaged with thermal test dies to record chip junction temperatures accurately. Small wind tunnels were used to determine package and heat sink thermal performance for various airflows. Larger wind tunnels were used to evaluate board airflow to give the board designers feedback on component placement by monitoring preheat conditions and flow obstructions. On printed circuit boards, external plane thermal dissipation pads were used where possible in lieu of adding heat sinks to some surface mount parts.

A full-scale system mockup was built. Various board models, air filters, EMI screens, and vents were tried to gather system airflow resistance data to determine the size and number of fans required. Various cooling schemes were evaluated by altering airflow direction and fan location. Pulling air down through the cabinet was found to provide uniform airflow across the cardcages while keeping the air filters clean by their high location. Having the fans low in the product and away from the vents kept noise sources farther away from operators and made servicing the fans easier.

The eleven dc fans in the fan tray have the ability to run at three different speeds: high, normal, or low. Seven fans run at low speed during startup and battery backup to keep the power use at a minimum while supplying sufficient cooling. All eleven fans run at normal speed while the system is up and running with the inlet air at or below 30°C. In this case the system meets the acoustic noise limit of 7.5 bels (A-weighted) sound power. The fans run at high speed while the system is up and running with the inlet air above 30°C or when the temperature rise through the processor memory bus cardcage exceeds 15°C.

At high speed, the fan tray has a volumetric airflow of approximately 1200 ft^3/min, which is designed to handle over six kilowatts of heat dissipation. This amount of power was considered early in the project when alternate chip technologies were being investigated. Therefore, the Model T500 has a cooling capacity of approximately one watt per square centimeter of floor space, a threefold increase over the high-end platform that the Model T500 is replacing, yet it is still air-cooled. The minimum air velocity is two meters per second in all of the processor memory bus slots and the typical air velocity is 3 m/s.

Because the processor memory bus cardcage contains high pressure drops and airflows, the board loading sequence is important, especially for the processor boards. Since the heat sinks are on the right side of the vertical processor boards, they are loaded sequentially from right to left. This ensures that air is channeled through the processor heat sinks instead of bypassing them in large unfilled portions of the cardcage.

Manufacturing

The fundamental strategy for manufacturing the HP 9000 Model T500 corporate business server was concurrent engineering, that is, development of both the computer and the technologies and processes to manufacture it at the same time. This resulted in a set of extensions to existing high-volume, cost-optimized production lines that allow sophisticated, performance enhancing features to be added to the corporate business server.

Cyanate Ester Board Material

Printed circuit boards based on cyanate ester chemistry (referred to as HT-2) have much better thermal, mechanical, and electrical performance than typical FR-4 substrates. These properties make HT-2 ideally suited for large printed circuit assemblies with intensive use of components with finely spaced leads, high-reliability applications, high-frequency applications, and applications with tight electrical tolerances.

More advanced printed circuit board designs tend to increase the aspect ratio of the board, or the ratio of the thickness of the board to the width of the vias for layer-to-layer connections. This is hazardous for FR-4 substrates because higher-aspect-ratio vias tend to be damaged in the thermal cycles of printed circuit assembly processes because of the expansion of the thickness of the boards in these cycles. The reliability of vias and through-hole connections (where the processor memory bus connector or VLSI pin-grid arrays are soldered to the board) is essential to the overall reliability, manufacturability, and repairability of the Model T500 memory board.

Because of their high glass transition temperature, HT-2 substrates are ideally suited to survive the stressful assembly and repair processes and to increase the yields within these processes. The glass transition temperature is the temperature at which the laminated fiberglass printed circuit board transitions from a solid to a pliable material. This is exceeded for FR-4 in the printed circuit assembly process, resulting in distortions of the boards. If no fine-pitch or extra-fine-pitch parts are used, the distortion for FR-4 is acceptable in the surface mount process. For large boards that use fine-pitch components, the surface mount processes tolerate less distortion. HT-2 has the advantage that it remains stable because it doesn't reach its glass transition temperature in the manufacturing process.

Printed Circuit Assembly and Test

Model T500 system requirements, through their impact on the memory board design, required development of significant new printed circuit assembly process capability. This process development effort began two years before volume shipments and identified the important areas for engineering effort. New technology introduced in printed circuit assembly included advanced reflow techniques. This is important because the total thermal mass of the components reflowed on the memory board is large, and because of the nature of the connector used for the processor memory bus. Special solder paste application methods were developed for the processor memory bus connector. This provides the assembly process with wide latitude for the connectors, pin-grid arrays, standard surface mount parts, and fine-pitch parts.

A key benefit of the cyanate ester choice for double-sided assemblies is reduced runout of the board, resulting in improved registration of the solder-paste stencil and components for higher yields of solder joints. In the double-sided surface mount process, the B side components are placed and reflowed before placement of the A side components. Since reflow for the B side is conducted at a temperature far above the glass transition temperature for standard FR-4 material, the boards would have been distorted in this step if FR-4 had been used. Thus FR-4 boards would have a higher failure rate on solder connections for the A side components.

Printed circuit board test was another area identified by the early concurrent engineering effort. Model T500 printed circuit assemblies are tested using a strategy extended from the HP 3070 board test system. Much of the test is conducted using leading-edge scan techniques. For example, because of trace length and capacitance, it was impossible to add test points between the processor memory bus drivers of the bus transceivers and the arbitration and address buffer and the resistor pack that connects to the bus without impacting performance. A scheme was devised using the scan port to activate each chip's drivers. The HP 3070 is set to apply a known current to each resistor and measure the voltage drop, from which the resistor's value and connectivity can be determined. There is no loss of test coverage for these fine-pitch parts and the scheme has the added benefit of verifying much of the chip's functionality. Because of the chip design lead time, HP's own scan port architecture (designed several years in advance of the IEEE 1149 standard for this type of test approach) is used and custom software tools were developed. Current chip designs contain the IEEE 1149 scan port which is directly supported by the HP 3070.

A major manufacturing challenge was the total number of nets and the board layout density found in the memory board. With 4273 nets, if normal HP 3070 design rules, which require one test point per net, were followed as much as 20% of the surface of the board would have been dedicated to test points. To solve this problem a scan-based approach is used on the nets where VLSI parts have scan ports. By using the scan ports and exercising some of the MSI part functionality, the number of nets that need test points is reduced to 2026.

This approach freed board space and allowed the needed density to be achieved. If this density had not been achieved, the alternative would have been to lower the capacity of each memory board, thereby lowering the overall system memory capacity.

Package Design Using 3D Solid Modeling

The industrial design and product design groups designed the HP 9000 Model T500 corporate business server package using the HP ME 30 solid modeling system. In the past, designs were drawn as 2D orthographic layouts. These layouts were then dimensioned and paper copies were given to the vendor for fabrication. Now, 3D bodies are sent directly to vendors via modem without having to dimension them. A 2D drawing is also sent to the vendor to provide a view of the part, usually isometric, and to call out notes and necessary secondary operations (plating, tolerances, cosmetic requirements, press-in fastener installations, etc.).

Using 3D solid modeling allowed the product design group to reduce design time, reduce part documentation time, and reduce design errors caused by using 2D layouts (with orthographic views, all three 2D views must be updated for a design change instead of a single 3D body). Additional benefits are faster turnaround on prototypes and an improved process for creating assembly documentation (isometric views of assembly positions are easily created by manipulating 3D bodies).

Eight engineers created approximately 150 sheet-metal parts, ten plastic parts, 25 cables, 15 miscellaneous parts, and many board components. Managing such a large mechanical assembly was initially thought to be too difficult. But having an organized file structure and 3D body placement strategy allowed the design team to work together efficiently. All engineers worked on their own assemblies, stored in separate write-protected directories, and were able to view adjoining assemblies for interface design.

The service processor presented two major challenges to make it fit both electrically and mechanically onto the HP 3070 test fixture. The total of 2312 nets on this board made it important to make all possible electrical pins of the test fixture available, which was difficult considering the large number of components. This problem was alleviated by careful layout of the service processor with the test fixture in mind. A custom fixture was designed to accommodate the board with its 2.5-inch bulkhead.

All of the boards and fixtures are designed to accommodate the transition to a no-clean process, which allows manufacturing of printed circuit assemblies without a chlorofluorocarbon (CFC) wash. This advanced work was driven by Hewlett-Packard's commitment to the total elimination of CFCs, which have been shown to destroy the ozone layer. The elimination of CFC use at HP was accomplished by May 15, 1993, more than two years ahead of the Montreal Protocol goal for an international ban on the use of these chemicals.

Mechanical and Final Assembly and Test

A key focus of concurrent design for manufacturability was the frame and cardcage design. Early effort by the design team and manufacturing went into detecting areas to improve the design for ease of assembly, to minimize the number and variety of fasteners, and to reduce the number of stocked items. This resulted in a set of features that include:

- Extensive use of captive fasteners, that is, fasteners that are preplaced in mechanical subassemblies. This reduces the number of individual mechanical parts to handle during assembly.
- A minimal set of unique fasteners with extensive use of Torx fasteners.

- One-direction fastening. Assemblers are not required to reach around or use awkward movements during assembly.
- A simplified assembly procedure. Only one piece to pick up and handle during any operation.
- Modularity. It is very easy to install or replace many components in the chassis without interference.
- Extensive use of high-density connectors for wiring harnesses. This reduces wiring time and errors. Point-to-point wiring is minimal.
- A robust cabinet and a very strong frame. The frame can survive shipping on its casters alone, and does not require a special pallet for most shipments.
- A refrigerator-sized cabinet that when fully loaded (approximately 360 kg) can still be moved easily by any operator or technician.

The Model T500 is designed with many inherent testability features, most of which are accessible using the system console. The system console is one of the most fundamental functions of the Model T500. It can be used in the earliest steps in bringing up and testing a newly assembled system. This permits extensive control and monitoring capability from a single communication point for manufacturing's automated test control host, and eliminates the need for many additional custom devices traditionally used for testing large computer systems. Many of the testability features benefit both manufacturing and customer support. The capabilities used for manufacturing test include the following:

- Monitor and change system parameters (such as secondary voltages or power system status) from the system console.
- Review from the console the system activity logs which track events that may indicate incorrect operation.
- Change self-test configuration. Select only the tests desired, or repeat tests to aid defect analysis.
- Access diagnostics through a LAN connection standard on all configurations of the system.
- Diagnose potential failure sources down to a specific integrated circuit.
- Use scan tools designed closely to manufacturing test specifications.

Acknowledgments

The authors would like to thank the following for their contributions to this article: Manfred Buchwald, Ken Chan, Ira Feldman, Dick Fowles, Chuen Hu, Ed Jacobs, Dong Nguyen, Phil Nielson, Nazeem Noordeen, Tom Wylegala, Syrus Ziai, and Steve Thomas for his photographic support. The authors also wish to thank the many individuals involved in making the HP corporate business server possible. This includes individuals from HP's mainline systems lab in Cupertino, California, the engineering systems lab in Fort Collins, Colorado, the IC design lab in Fort Collins, Colorado, the Böblingen Printed Circuit Manufacturing Operation in Böblingen, Germany, the Networked Computer Manufacturing Operation in Roseville, California, and the Cupertino open systems lab in Cupertino, California.

References

1. R.B. Lee, "Precision Architecture," *IEEE Computer,* Vol. 22, January 1989.
2. D. Tanksalvala, et al, "A 90-MHz CMOS RISC CPU Designed for Sustained Performance," *ISSCC Digest of Technical Papers,* February 1990, pp. 52-53.
3. J. Lotz, B. Miller, E. DeLano, J. Lamb, M. Forsyth, and T. Hotchkiss, "A CMOS RISC CPU Designed for Sustained High Performance on Large Applications," *IEEE Journal of Solid-State Circuits,* October 1990, pp. 1190-1198.
4. P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer,* Vol. 23, no. 6, June 1990, pp. 12-25.
5. *TPC Benchmark C Full Disclosure Report: HP 9000 Corporate Business Server T500, Using HP-UX 10.0 and INFORMIX OnLine 5.02,* Hewlett-Packard Company, November 1993.
6. *TPC Benchmark A Full Disclosure Report: HP 9000 Corporate Business Server 890, Using HP-UX and ORACLE7 Client/Server Configuration,* Hewlett-Packard Company, publication no. 5091-6844E, February 1993.
7. Standard Performance Evaluation Corporation, Results Summaries, *SPEC Newsletter,* Volume 5, Issue 4, December 1993.
8. E. Delano, W. Walker, J. Yetter, and M. Forsyth, "A High-Speed Superscalar PA-RISC Processor," *COMPCON Spring '92 Digest of Technical Papers,* February 1992.
9. R.C. Brockmann, W.S. Jaffe, and W.R. Bryg, *Flexible N-Way Memory Interleaving,* U.S. Patent Application, February 1991.
10. K. Chan, et al, "Multiprocessor Features of the HP Corporate Business Servers," *COMPCON Spring '93 Digest of Technical Papers,* February 1993.

PA-RISC Symmetric Multiprocessing in Midrange Servers

By making a series of simplifying assumptions and concentrating on basic functionality, the performance advantages of PA-RISC symmetric multiprocessing using the HP PA 7100 processor chip were made available to the midrange HP 9000 and HP 3000 multiuser system customers.

by Kirk M. Bresniker

The HP 9000 G-, H-, and I-class and HP 3000 Series 98x servers were first introduced in the last quarter of 1990. Over the lifetime of these systems almost continual advances in performance were offered through increases in cache sizes and processor speed. However, because of design constraints present in these low-cost systems, the limits of uniprocessor performance were being reached.

At the same time, the HP PA 7100 processor chip was being developed. Its more advanced pipeline and superscalar features promised higher uniprocessor performance. Advances in process technology and physical design also promised higher processor frequencies.

Part of the definition of the PA 7100 is a functional block that allows two PA 7100 processors to share a memory and I/O infrastructure originally designed for a single processor. This functional block provides all the necessary circuitry for coherent processor communication. No other system hardware resources are necessary. This feature of the PA 7100 processor made it technically feasible to create a very low-cost two-way symmetric multiprocessing processor board for the HP 9000 and HP 3000 midrange servers. However, significant design trade-offs had to be made to create a product in the time frame necessary.

This article describes the design of this new processor board, which is used in the HP 9000 Models G70, H70, and I70 servers. The HP 3000 Series 987/200 business computer is based on the same processor board.

Design Goals

The design goal of the system was to provide the advantages of symmetric multiprocessing in the midrange servers both to new customers in the form of a fully integrated server and to existing customers in the form of a processor board upgrade. The only constraint was that existing memory, I/O cards, and sheet metal had to be used. Everything else was open to possible change. However, a strong restoring force was provided by the need to minimize time to market and the very real staffing constraints. There simply weren't time or resources to enable us to provide all the features associated with symmetric multiprocessing. The decision was made to make the performance advantages of symmetric multiprocessing the primary design goal for the midrange servers.

Development History

The I-class server was chosen as the initial development platform for the PA 7100 processor. An I-class processor board was developed that accepts a PA 7100 module consisting of the processor package and high-speed static RAMs. In addition, an extender board was developed that allows two PA 7100 modules to be connected to the I-class processor board. This four-board assembly, which was the first prototype of the eventual design, booted and was fully functional within five months of the initial PA 7100 uniprocessor turn-on. This short time period allowed all the basic operating system changes and performance measurements to be made at the same time as the uniprocessor work was being done, by the same design team, with only a small incremental effort.

At this point, the efforts of the design team were centered on introducing the PA 7100 uniprocessor servers. However, since the initial performance measurements of the symmetric multiprocessing prototype were so encouraging, the team continued to refine and develop the initial prototype into a manufacturable product.

The first decision of the design team was to implement the design using 1M-byte instruction and data caches, a fourfold increase over the initial PA 7100 designs. This decision was driven by the initial performance measurements made on prototypes, which showed that the larger caches optimized the utilization of the shared processor memory bus. The same measurements also showed that the most desirable performance levels would require the design to match the previous processor frequency of 96 MHz. This would be the first of the large-cache, high-speed designs for the PA 7100 processor, and would therefore carry considerable design risk.

The next decision was to implement the design not with modules, but as a single board. This was done to lower the cost and technology risk of the design. The shared processor memory bus would be twice as long as in previous designs, but it would not have to bear the additional signal integrity burden of two module connector loads. This was the first of the simplifying assumptions, but it led to several key others.

A great deal of the complexity in symmetric multiprocessing systems arises not just from the problems of maintaining the

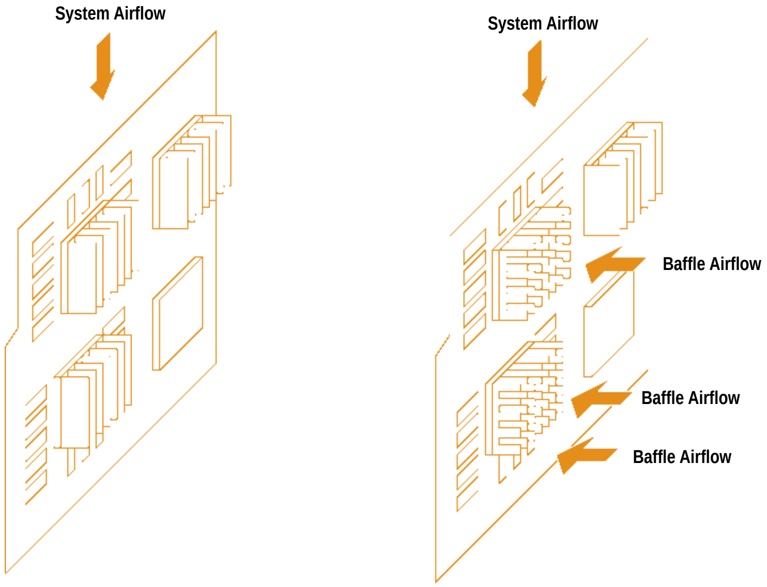


Fig. 1. On the left is the unmodified airflow pattern showing the second processor in the thermal shadow of the first. On the right is the revised airflow pattern showing the impingement cooling provided by the baffle fan.

processors during normal operation, but from handling special operating conditions like failures or booting. Since in this case both processors are always installed, one processor is designated as the "monarch" and is allocated special responsibilities. The second processor is designated as the "serf," and is not allocated any special responsibilities. This obviates the need for a complex method of determining which processor should maintain control during exceptional circumstances. Also, since both processors are on the same board and cannot be replaced independently, it was decided that if one processor should fail, the other would not continue to operate. This removes an entire class of complex interactions that would have had to be discovered, handled, and tested, considerably shortening the firmware development life cycle.

One negative implication of the single-board solution was that one processor was in the direct airflow path of the other (see Fig. 1). This meant that a new solution for cooling had to be devised, but in such a way that the upgrade to the new design would not impact the existing sheet metal. A passive solution of diverting the airflow using air baffles did not prove to be effective enough, so the mechanical design team devised an active solution. A forced-air baffle was devised that is essentially a box occupying the airflow volume next to the processor board. It has three openings centered above the processors and the worst-case cache components. The box is pressurized by a miniature fan. This causes air to impinge directly on the critical components without disturbing the airflow to the rest of the processor board. Since the primary airflow is now normal to the processor board, a new heat sink consisting of a grid of pins was devised to allow the impinging air to cool the processors most efficiently.

One drawback of this active airflow solution is that it relies so heavily on the miniature fan to maintain the processor temperature in a safe range. Of all component classes used in these systems, fans have some of the higher failure rates. Since so much of the air volume next to the processor board is committed to the forced-air baffle, failure of the forced-air baffle fan can cause permanent damage to the processors if not detected in time. In fact, the overheating of the processors was measured to be so rapid in the event of the baffle fan failure that the existing overtemperature protection could not be activated quickly enough. For this reason, the fan is continuously monitored. If the fan stops spinning or rotates slower than a preset limit, the system power supplies are shut down immediately. In addition to providing maximum protection to the processors, this solution also removes the need to burden the software and firmware development with status checking routines.

All of these decisions were made in the background, while the uniprocessor design was being readied for release. In fact, some of the impetus for making the simplifications was the lack of time. However, it was clear that the desire for the system was strong enough for the team to continue. Within one week of the release of the final revision of the uniprocessor system, the initial revision of the multiprocessor processor board was also released. This functional prototype proved to be extremely stable, with no hardware failures reported during the design phase.

Verification

It was at this point that the electrical verification of the design began, and with it the challenging phase of the project as well. The design risks of the large, high-speed caches imagined early on turned out to be all too real. The most problematic aspect of the cache design is that the read access budget for the cache access is one and one half clock cycles (15.6 ns, assuming 96-MHz operation). During that time, the address must be driven to the SRAMs, the SRAMs must access the data, and the data must be driven back to the processor. Current SRAM technology consumes almost 60% of the read budget in internal access time. This budget needs to be maintained over all possible operating conditions, and a single fault can cause either a reload (in the case of instructions) or a system panic and shutdown (in the case of data). The unique problem with this design was that caches this large had never before been run with the PA 7100 processor.

The test methodology used was to run tests tailored to stress the caches while varying the system voltage, temperature, and frequency. Although functional testing at normal conditions had yielded no failures, the initial cache design quickly

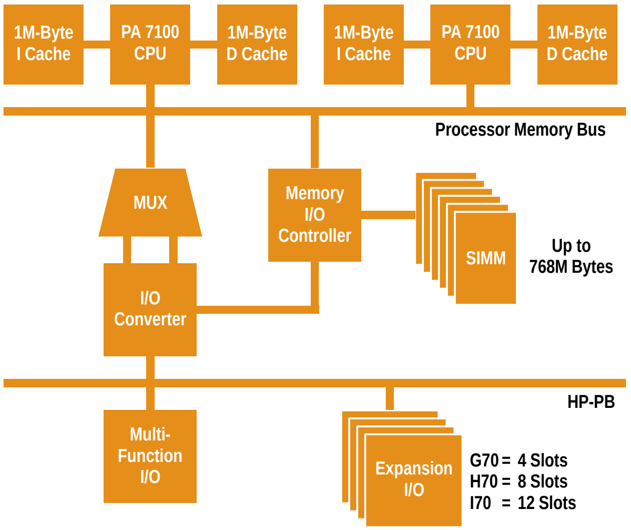


Fig. 2. Block diagram of the HP 9000 Model I70 computer system.

succumbed to the pressures of this type of electrical verification. Analysis of the failures indicated that the read budget was being violated at the combined extremes of low voltage, high temperature, and high frequency. The 1M-byte SRAMs had higher capacitive loads and were physically larger than their lower-density counterparts. This greatly increased the address drive time. The team did not have recourse to faster high-density SRAMs from any vendor, and caches built out of faster lower-density SRAMs would not have provided the symmetrical multiprocessing performance we desired.

What followed was an exhaustive analysis by all three contributors to the design: the PA 7100 design team, the board design team, and the SRAM vendor design teams. Each team worked at pulling fractions of nanoseconds out of the read access. The board design team experimented with termination designs and new layouts to improve address drive time. The PA 7100 team pushed their chip faster to increase the read time budget. They also identified which critical signals had to be faster than all the rest and simulated the board team's changes. The SRAM vendor design teams pushed their processes to achieve faster components. All three teams pushed their designs to the limits, and it took contributions from all three teams to succeed. In the end, it took over six months of constant design refinement and testing to achieve the final result, a design that meets the team's initial electrical verification requirements. This turned out to be the only significant electrical design problem that the processor board team had to solve.

While the board design team worked out the electrical design issues, a separate team was formed to verify the multiprocessing functionality of the PA 7100 processor. This formal verification was the last step in the development cycle for the systems.

System Overview

A block diagram of the Model I70 system appears in Fig. 2. Both PA 7100 CPUs are configured with 1M bytes of instruction cache and 1M bytes of data cache. The processors run at a speed of 96 MHz. The shared processor memory bus is operated at a fixed ratio of 3:2 with respect to the processors, or 64 MHz, and connects the processors to the single memory and I/O controller. The memory and I/O controller interfaces to a maximum of 768M bytes of error corrected memory. The I/O adapter connects a demultiplexed version of the shared processor memory bus to a four-slot (Model G70), eight-slot (Model H70), or twelve-slot (Model I70) HP-PB (Hewlett-Packard Precision Bus) I/O bus.

In addition to the processor board, the base system consists of the HP-PB backplane, a memory extender, a fan baffle, and a multifunction I/O card.

System Specifications

The following specifications are for the 12-slot Model I70 server.

Processors	2 PA 7100 superscalar processors with integrated floating-point unit
Cache	1M-byte instruction cache per processor. 1M-byte data cache per processor
Processor Clock	96 MHz
System Clock	64 MHz
Maximum Memory	768M bytes
I/O Bus	1 12-slot HP-PB
Maximum Integrated Storage	6G bytes
Maximum External Storage	228G bytes
Maximum LANs	7
Maximum Users	3500

Summary

The success of bringing PA-RISC symmetric multiprocessing to the HP 9000 and HP 3000 midrange servers was the result of implementing simplified symmetric multiprocessing functionality. The PA 7100 team integrated all the functionality for two-way symmetric multiprocessing into their design. The system design team followed their lead by creating a system around the two processors that includes only the core hardware and firmware functionality absolutely necessary for operation.

Acknowledgments

The author wishes to acknowledge the members of the hardware and firmware development team under Trish Brown and Ria Lobato: Jim Socha, Janey McGuire, Cindy Vreeland, and Robert Dobbs. Special thanks go to Lin Nease, who implemented the original firmware and operating system changes. The operating system work was done by Anastasia Alexander and Steve Schaniel. Also, thanks go to Jeff Yetter's PA 7100 team, especially Greg Averill, implementor of the processor memory bus sharing functionality, and Tom Asprey, Bill Weiner, and Tony Lelm for their help in the electrical verification of the cache design. Also instrumental in the verification of the cache design were Bill Hudson and the engineers of the Motorola fast SRAM group. The success of the multiprocessor functional verification is the result of the efforts of the multiprocessor verification team of Akshya Prakash. The excellent work of both the electrical and multiprocessor verification teams was evidenced when the Model H70 was chosen as the minicomputer product of the year in the 1993 *VAR Business* reader survey. When asked to explain the nomination, one reader quipped, "It doesn't crash."

SoftBench Message Connector: Customizing Software Development Tool Interactions

Software developers using the SoftBench Framework can customize their tool interaction environments to meet their individual needs, in seconds, by pointing and clicking. Tool interaction branching and chaining are supported. No user training is required.

by Joseph J. Courant

SoftBench Message Connector is the user tool interaction facility of the SoftBench Framework, HP's open integration software framework. Message Connector allows users to connect any tool that supports SoftBench Framework messaging to any other tools that support SoftBench Framework messaging without having to understand the underlying messaging scheme. Users of the framework can easily customize their tool interaction environments to meet their individual needs, in literally seconds, by simply pointing and clicking.

People familiar with the term SoftBench may know it under one or both of its two identities. The term SoftBench usually refers to a software construction toolset.[1] The term SoftBench Framework refers to an open integration software framework often used to develop custom environments.[2] People familiar with SoftBench the toolset should know that underlying the toolset is the SoftBench Framework.

Message Connector can be used to establish connections between any SoftBench tools without understanding the underlying framework. The editor can be connected to the builder which can be connected to the mail facility and the debugger, and so on. Message Connector does not care what tools will be connected, as long as those tools have a SoftBench Framework message interface. The message interface is added by using the SoftBench Encapsulator,[3] which allows users to attach messages to the functions of most tools. Message Connector uses the message interface directly and without modification. To date, over seventy known software tools from a wide variety of companies have a SoftBench message interface. It is also estimated that a much larger number of unknown tools have a SoftBench message interface. Users of the SoftBench Framework can now treat tools as components of a personal work environment that is tailored specifically by them and only takes minutes to construct.

Tools as Components

What does it mean to treat tools as components? To treat a tool as a component means that the tool provides some functionality that is part of a larger task. It is unproductive to force tool users to interact with several individual tools to accomplish a single task, but no tool vendor is able to predict all of the possible ways in which a tool's functionality will be used. Using Message Connector, several tools can be connected together such that they interact with each other automatically. This automatic interaction allows the user to focus on the task at hand, not on the tools used to accomplish the task.

A simple but powerful example is detecting spelling errors in a document, text file, mail, or any other text created by a user. The task is to create text free of spelling errors. The tools involved are a text editor and a spell checker. In traditional tool use, the editor is used to create the text and then the spell checker is used to check the text. In simple notes or files the text is often not checked for errors because it requires interacting with another tool, which for simple text is not worth the effort. When treating tools as components the user simply edits and saves text and the spell checker checks the text automatically, only making its presence known when errors exist. Note that in traditional tool use there is one task but two required tool interactions. In the component use model, there is one task and one required tool interaction (see Fig. 1).

Using Message Connector, a user can establish that when the editor saves a file, the spell checker will then check that specific file. This is accomplished as follows:

1. Request that Message Connector create a new routine (routine is the name given to any WHEN/THEN tool interaction).
2. Select the WHEN: tool to trigger an action (editor).
3. Select the specific function of the WHEN: tool that will trigger the action (file saved).
4. Select the THEN: tool to respond to the action (spell checker).
5. Select the specific function that will respond (check file).
6. Change the WHEN: and THEN: file fields to specify that the file saved will be the file checked.
7. Save the routine (routines are persistent files allowing tool interactions to be retained and turned on and off as desired).
8. Enable the routine.

Now any time the editor saves a file, that file will automatically be spell checked. The focus of creating text free of spelling errors is now the editor alone. The spell checking is driven by editor events, not by the user.

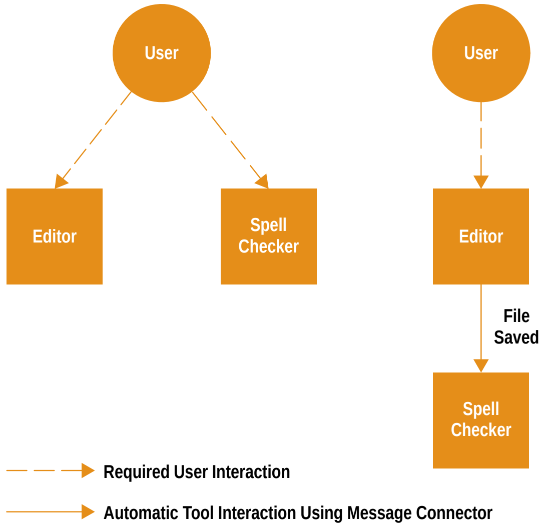

User
Editor
Spell Checker
User
Editor
File Saved
Spell Checker
Required User Interaction
Automatic Tool Interaction Using Message Connector

Fig. 1. Traditional tool use (left) compared with tools as task components (right). Using the SoftBench Message Connector, the user can set up a routine so that whenever a file is saved by the editor it is automatically spell checked. The spell checker does not have to be explicitly invoked by the user.

Tool Interaction Branching

While the above example is very simple, it applies equally well to any number of tool interactions. It is also possible to create branching of interaction based upon the success or failure of a specific tool to perform a specific function (see Fig. 2). For example, when the build tool creates a new executable program then display, load, and execute the new program within the debugger; when the build tool fails to create a new executable then go to the line in the editor where the failure occurred.

Interaction Chaining

It is possible to define interactions based upon a specific file type, and it is also possible to chain the interactions (see Fig. 3). As an example, when the editor saves a text file then spell check that file; when the editor saves a source code file then perform a complexity analysis upon that file; when a complexity analysis is performed on a file and there are no functions that exceed a given complexity threshold then build the file; when the complexity is too high, go to the function in the editor that exceeds the given complexity threshold; when the build tool creates a new executable program then display, load, and execute the new program within the debugger, reload the new executable into the static analysis tool, and save a version of the source file; when the build tool fails to create a new executable, then go to the line in the editor where the failure occurred. This example of interaction chaining allows the user to focus on the task of creating defect-free text and source files. The user's focus is on the editor and all other tools required to verify error-free files are driven automatically by editor events, not by the user. The tools have become components of a user task.

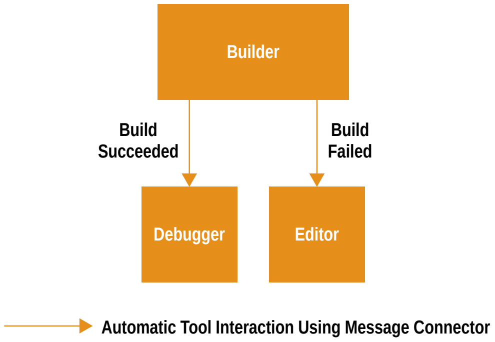


Fig. 2. Message Connector supports tool interaction branching. A different tool is invoked automatically depending on the result of a previous operation.

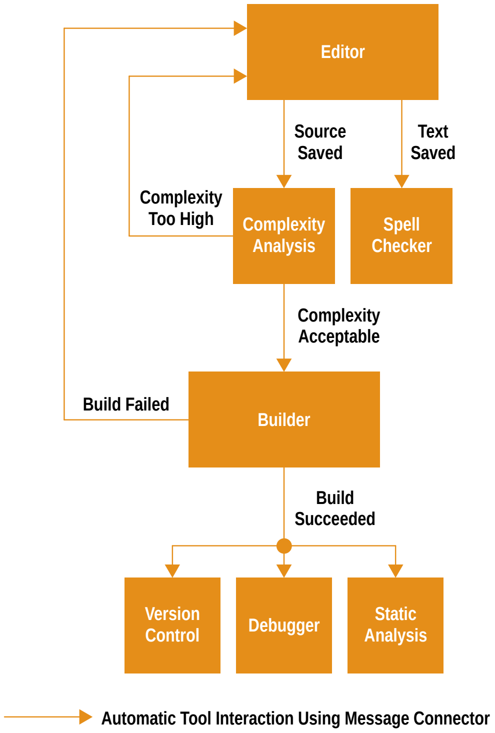


Fig. 3. Tool interaction chaining.

Message Connector Architecture

The architecture of Message Connector follows the component model of use encouraged by Message Connector. As shown in Fig. 4, Message Connector is a set of three separate components. Each component is responsible for a separate function and works with the other components through the SoftBench Framework messaging system. The routine manager provides the ability to enable, disable, organize, and generally manage the routines. The routine editor's function is routine creation and editing. The routine engine's function is to activate and execute routines.

The importance of this architecture is that it allows Message Connector, the tool that allows other tools to be treated as components, to be treated as a set of components. This allows the user, for example, to request that the routine engine enable or disable another routine within a routine. It allows the user to run a set of routines using the routine engine without a user interface. It allows the user to request that the routine engine automatically enable any routine saved by the routine editor. Many other examples of the advantages of the architecture can be given.

The routine manager simply gives the user a graphical method of managing routines. When analyzing the tasks a

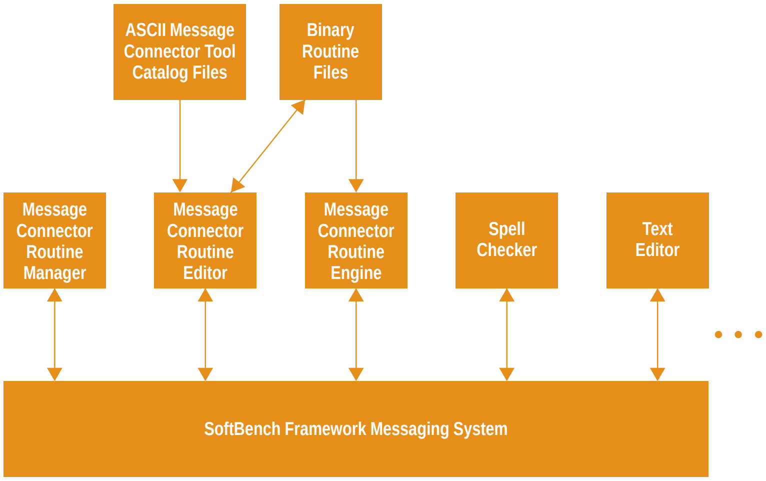

ASCII Message Connector Tool Catalog Files
Binary Routine Files
Message Connector Routine Manager
Message Connector Routine Editor
Message Connector Routine Engine
Spell Checker
Text Editor
• • •
SoftBench Framework Messaging System

Fig. 4. SoftBench Message Connector architecture. The three major Message Connector modules—the routine manager, the routine editor, and the routine engine—are treated as components like the tools.

Message Connector user would perform, it was concluded that the routine manager would be in the user's environment most of the time. It was also concluded that the routine manager would be an icon most of the time. As a result, the design goal for the routine manager was to occupy as little screen space, memory, and process space as possible. As designed and implemented, a large portion of the routine manager's user interface simply sends a message to the SoftBench Framework requesting that a service be performed. As an example, the Enable and Disable command buttons simply send a message requesting that the routine engine enable or disable the selected routine. The routine manager was designed, implemented, and tested before the implementation of the routine editor and the routine engine.

The routine editor proved to be very challenging. The Message Connector project goal stated that, "Message Connector will provide SoftBench Framework value to all levels of end users in minutes." While a simple statement, the implications were very powerful. "All levels of end users" implied that whatever the editor did, displaying the underlying raw framework would never meet the goal. All information would have to be highly abstracted, and yet raw information must be generated and could not be lost. "All levels of end users" also implied that any user could add messaging tools to the control of Message Connector, so Message Connector could not have a static view of the framework and its current tools. "In minutes" implied that there would be no need to read a manual on a specific tool's message interface and format to access the tool's functionality. It also implied that the routine editor, tool list, and tool function lists must be localizable by the user without disturbing the required raw framework information. "In minutes" also implied that there would be no writing of code to connect tools.

The routine editor underwent sixty paper prototype revisions, eighteen code revisions, and countless formal and informal cognitive tests with users ranging from administrative assistants to tenured code development engineers. It is ironic that one result of focusing a major portion of the project team's effort on the routine editor has been that various people involved with promoting the product have complained that it is too easy to use. Apparently people expect integration to be difficult, and without a demonstration, potential customers question the integrity of the person describing Message Connector. When someone is told that there is a tool that can connect other disparate tools that have no knowledge of each other, in millions of possible ways, in seconds, without writing code, it is rather hard to believe.

The routine engine turned out to be an object-oriented wonder. The routine engine must be very fast. It stores, deciphers, matches, and substitutes portions of framework messages, it receives and responds to a rapid succession of a large number of trigger messages, and it accommodates future enhancements. The routine engine is the brain, heart, and soul of Message Connector and is completely invisible.

Example Revisited

Walking through the eight steps in the simple editor/spell checker example above will show the interaction within and between each of the Message Connector components.

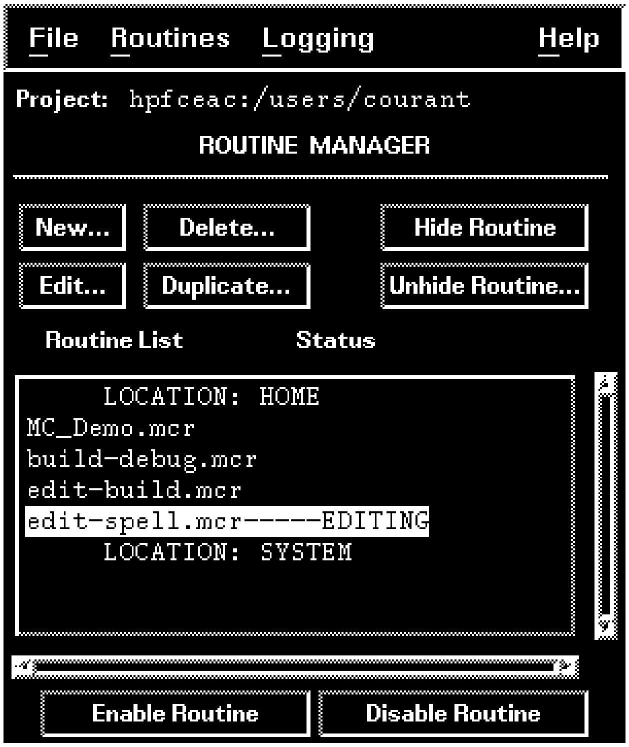
1. Request that Message Connector create a new routine.

This step is accomplished using the routine manager (see Fig. 5). The routine manager's task is to prompt the user for a routine name, ensure that the name has the proper file extension (.mcr), and then simply send a request to the message server to edit the named routine. The routine manager's role is largely coordination. It has no intimate knowledge of the routine editor. After sending the request to the message server to edit the named routine, the routine manager will await a notification from the message server of whether the edit was a success or a failure. The routine manager then posts the status of the request.

A separate routine editor is started for each routine edit request received by the message server. When the routine manager sends a request to the message server to edit a routine, the message server starts a routine editor and the routine editor initializes itself and sends a notification of success or failure back to the message server. Fig. 6 shows a typical routine editor screen.

2. Select the WHEN: tool to trigger an action.

In the case of creating a new routine, there is no routine to load into the editor and therefore the WHEN: and THEN: fields are displayed empty. The routine editor searches for and displays all possible tools available for Message Connector to manipulate. It is important that Message Connector is actually searching for Message Connector tool catalog files, not the tools themselves. For each file found, the file name is displayed as a tool in the routine editor.



Project: hpfceac:/users/courant

ROUTINE MANAGER

Routine List Status

```
     LOCATION: HOME
MC_Demo.mcr
build-debug.mcr
edit-build.mcr
edit-spell.mcr-----EDITING
     LOCATION: SYSTEM
```

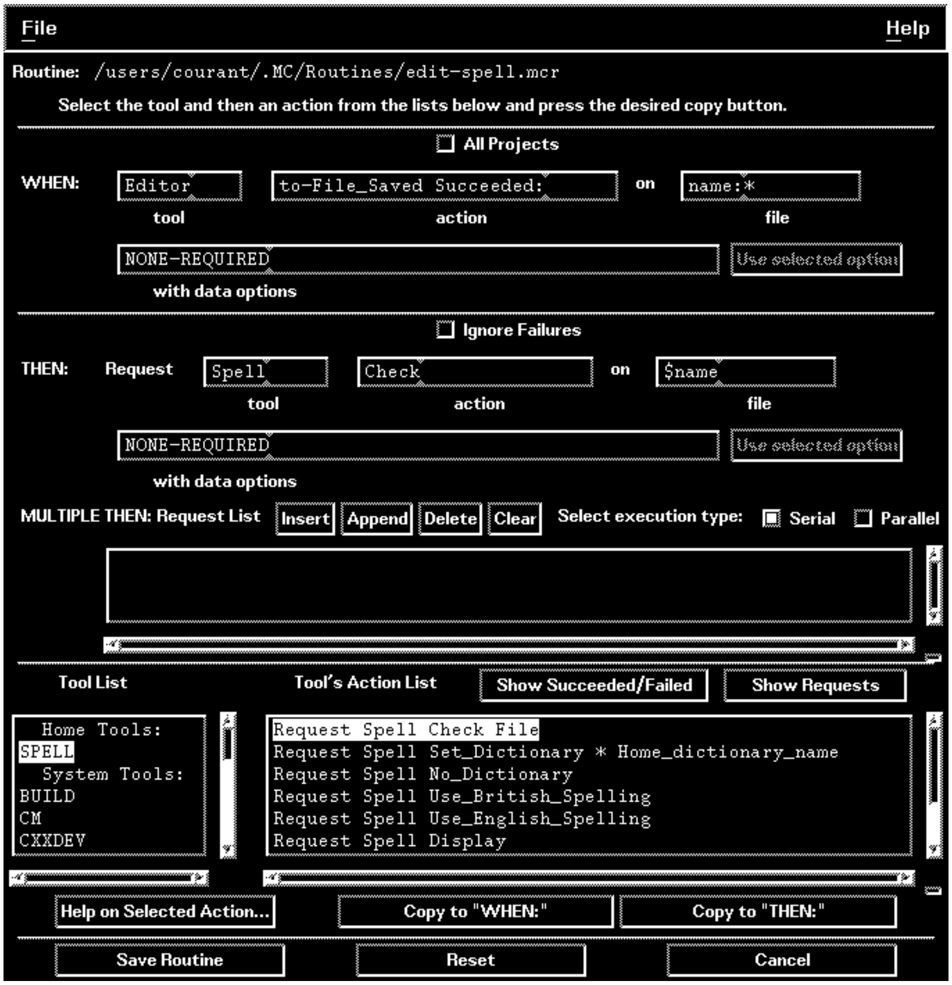
Fig. 5. Typical routine manager screen.

The Message Connector tool catalog files contain three important pieces of information. The catalog files are ASCII files that contain the raw messages required to access the functions of the tool being cataloged. The catalog files also contain the abstractions of the raw messages (these are displayed to the user, not the raw messages) and any message help that may be required by a user. For most tools, the catalog file is provided for the user by the person who added the message interface. If the catalog file does not exist for a particular tool, it can be created using that tool's message interface documentation. The catalog does not have to be created by the tool provider. The catalog files can also be edited by the user to change the abstraction displayed or to hide some of the seldom used functions.

When the user selects a tool from the Message Connector routine editor tool list, the routine editor goes out and parses the tool's catalog file for all applicable message abstractions and displays those abstractions.

3. Select the specific function of the WHEN: *tool that will trigger the action.*

When a user selects a WHEN: function and copies that function to the WHEN: statement, the routine editor reads the function's raw message and the abstraction of the raw message. Only the abstraction is displayed to the user, but both the raw message and the abstraction are temporarily preserved until the user saves the routine.



Routine: /users/courant/.MC/Routines/edit-spell.mcr

Select the tool and then an action from the lists below and press the desired copy button.

□ All Projects

WHEN: tool: Editor action: to-File_Saved Succeeded: on file: name:*

with data options: NONE-REQUIRED

□ Ignore Failures

THEN: Request tool: Spell action: Check on file: $name

with data options: NONE-REQUIRED

MULTIPLE THEN: Request List

Select execution type: ▣ Serial □ Parallel

Tool List

```
  Home Tools:
SPELL
  System Tools:
BUILD
CM
CXXDEV
```

Tool's Action List

```
Request Spell Check File
Request Spell Set_Dictionary * Home_dictionary_name
Request Spell No_Dictionary
Request Spell Use_British_Spelling
Request Spell Use_English_Spelling
Request Spell Display
```

Fig. 6. Typical routine editor screen.

4. Select the THEN: tool to respond to the action.

5. Select the specific function that will respond.

These steps are similar to the WHEN: steps.

6. Change the WHEN: and THEN: file fields.

This simply allows the user to change the values displayed on the screen. For these values, what is seen on the screen is what will be used when the user selects Save Routine.

7. Save the routine.

This step takes all of the raw messages, the message abstractions, and the screen values and assembles them into an internal routine file format which both the routine editor and the routine engine are able to read. The routine editor then writes out a binary data file into the routine file being edited and then quits.

8. Enable the routine.

This step is driven by the routine manager, but is performed by the routine engine. The user selects the routine of interest, then selects the Enable Routine button on the routine manager. Again, the routine manager's primary role is coordination. When the user selects the Enable Routine button, the routine manager simply finds the routine selected and sends a request to the message server to enable the named routine. The routine engine receives the enable request from the message server and reads the named routine. After reading the routine, the routine engine establishes the WHEN: message connection to the message server. This WHEN: connection is as general as required. If the user uses any wildcards in the WHEN: statement, the routine engine will establish a general WHEN: message connection and then wait until the message server forwards a message that matches the routine engine's message connection. If the message server forwards a matching message, the routine engine sends a request for each of the THEN: statements to the message server.

Development Process

Message Connector's transformation from a concept to a product was a wonderful challenge. The two most important elements of this transformation were a cross-functional team and complete project traceability. A decision was made before the first project meeting to assemble a cross-functional team immediately. To make the team effective, all members were considered equal in all team activity. It was made clear that the success or failure of the project was the success or failure of the entire team. This turned out to be the most important decision of the Message Connector project. The team consisted of people from human factors, learning products, product marketing, research and development, promotional marketing, and technical customer support. Most of the team members only spent a portion of their time on the Message Connector project. However, a smaller group of full-time people could never have substituted for Message Connector's cross-functional team. The collective knowledge of the team covered every aspect of product requirements, design, development, delivery, training, and promotion. During the entire life of the project nothing was forgotten and there were no surprises, with the exception of a standing ovation following a demonstration at sales training. The team worked so well that it guided and corrected itself at every juncture of the project.

One critical reason the team worked so well was the second most important element of the project—complete project traceability. There was not a single element of the project that could not be directly traced back to the project goal. This traceability provided excellent communication and direction for each team member. In the first two intense weeks of the project, the team met twice per day, one hour per meeting. These meetings derived the project goal, objectives (subgoals by team definition), and requirements. The rule of these meetings was simple: while in this portion of the project no new level of detail was attempted until the current level was fully defined, understood, and challenged by all members. As each new level of detail was defined, one criterion was that it must be directly derived from the level above—again, complete project traceability. The project goal was then posted in every team member's office to provide a constant reminder to make the correct trade-offs when working on Message Connector. This amount of time and traceability seemed excessive to some people outside of the team, but it proved to be extremely productive. All of the team members knew exactly what they were doing, what others were doing, and why they were doing it throughout the life of the project.

The project goal was made easy to remember, but was extremely challenging: "Message Connector will provide SoftBench Framework value to all levels of end users in minutes." At first glance, this seems very simple. Breaking the goal apart, there are three separate, very challenging pieces to the goal: "SoftBench Framework value," "all levels of end users," and "in minutes." As an example of the challenge, let's look more closely at the "in minutes" portion of the project goal. "In minutes" means that there is a requirement that the user find value in literally minutes using a new product that uses a rather complex framework and a large number of unknown tools that perform an unknown set of functionality. How would Message Connector provide all of this information without requiring the user to refer to any documentation? "In minutes" made a very dramatic impact on the user interface, user documentation, and user training (no training is required). These three pieces of the goal also provided the grounds for the project objectives. The project objectives then provided the basis for the project and product requirements. At each new level of detail it was reassuring to the team that there was no effort expended that did not directly trace back to the project goal. The team ownership, motivation, creativity, and productivity proved to be extremely high.

Conclusion

Using Message Connector, users of the SoftBench Framework can easily customize their tool interaction environment to treat their tools as components of a task, in literally seconds, by simply pointing and clicking. This was all made possible by immediately establishing a cross-functional team to own the project and requiring complete project traceability. An interesting fact is that early users of Message Connector developed two new components that are separate from the Message Connector product but are now shipped with it. One component (named Softshell) executes any specified

UNIX* command using messaging and can return the output of the command in a message. This allows a Message Connector user to execute UNIX commands directly as a result of an event of any tool. For example, when the user requests the editor to edit a file, if the file is read-only then execute the UNIX command to give the user write access. The second component (named XtoBMS) converts X Windows events into messages that Message Connector can use to request functionality from any component automatically. This means that when any tool maps a window to the screen, the user environment can respond with any action the user defines. This has been used extensively in process management tools so that the appearance of a tool on the screen causes the process tool to change the task a user is currently performing.

Acknowledgments

The transformation of Message Connector from concept to product was the result of a very strong team effort. While there were many people involved with the project at various points, the core Message Connector team consisted of Alan Klein representing learning products, Jan Ryles representing human factors, Byron Jenings representing research and development, Gary Thalman representing product marketing, Carol Gessner and Wayne Cook representing technical customer support, Dave Willis and Tim Tillson representing project management, and the author representing research and development. The cross-functional team approach caused roles and responsibilities to become pleasantly blurred. The entire team is the parent of Message Connector.

References

1. C. Gerety, "A New Generation of Software Development Tools," *Hewlett-Packard Journal,* Vol. 41, no. 3, June 1990, pp. 48-58.

2. M.R. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *ibid,* pp. 36-47.

3. B.D. Fromme, "HP Encapsulator: Bridging the Generation Gap," *ibid,* pp. 59-68.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Six-Sigma Software Using Cleanroom Software Engineering Techniques

Virtually defect-free software can be generated at high productivity levels by applying to software development the same process discipline used in integrated circuit manufacturing.

by Grant E. Head

In the late 1980s, Motorola Inc. instituted its well-known six-sigma program.[1] This program replaced the "Zero Defects" slogan of the early '80s and allowed Motorola to win the first Malcolm Baldridge award for quality in 1988. Since then, many other companies have initiated six-sigma programs.[2]

The six-sigma program is based on the principle that long-term reliability requires a greater design margin (a more robust design) so that the product can endure the stress of use without failing. The measure for determining the robustness of a design is based on the standard deviation, or sigma, found in a standard normal distribution. This measure is called a capability index (C_p), which is defined as the ratio of the maximum allowable design tolerance limits to the traditional ±3-sigma tolerance limits. Thus, for a six-sigma design limit $C_p = 2$.

To illustrate six-sigma capability, consider a manufacturing process in which a thin film of gold must be vapor-deposited on a silicon substrate. Suppose that the target thickness of this film is 250 angstroms and that as little as 220 angstroms or as much as 280 angstroms is satisfactory. If as shown in Fig. 1 the ±30-angstrom design limits correspond to the six-sigma points of the normal distribution, only one chip in a billion will be produced with a film that is either too thin or too thick.

In any practical process, the position of the mean will vary. It is generally assumed that this variation is about ±1.5 sigma. With this shift in the mean a six-sigma design would produce 3.4 parts per million defective. This is considered to be satisfactory and is becoming accepted as a quality standard. Table I lists the defective parts per million (ppm) possible for different sigma values.

At first the six-sigma measure was applied only to hardware reliability and manufacturing processes. It was subsequently recognized that it could also be applied to software quality. A number of software development methodologies have been shown to produce six-sigma quality software. Possibly the methodology that is the easiest to implement and is the most repeatable is a technique called cleanroom software engineering, which was developed at IBM Corporation's Federal Systems Division during the early 1980s.[3] We applied this methodology in a limited way in a typical HP environment and achieved remarkable results.

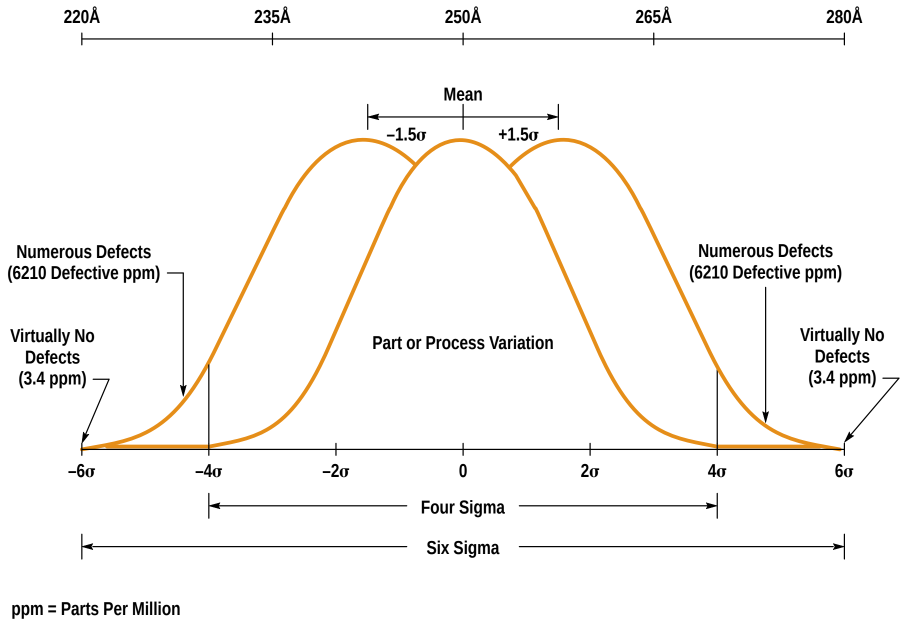


Fig. 1. An illustration of a six-sigma design specification. A design specification of ±30 angstroms corresponds to a ±6-sigma design. Also shown is the ±1.5σ variation from the mean as a result of variations in the process.

Table I
Defective ppm for Different Sigma Values

Sigma	ppm
1	697,700
2	308,733
3	66,803
4	6,210
5	233
6	3.4
7	0.019

When applied to software, the standard unit of measure is called *use,* and six-sigma in this context means fewer than 3.4 failures (deviations from specifications) per million uses. A use is generally defined to be something small such as the single transaction of entering an order or command line. This is admittedly a rather murky definition, but murkiness is not considered to be significant. Six-sigma is a very stringent reliability standard and is difficult to measure. If it is achieved, the user sees virtually no defects at all, and the actual definition of a use then becomes academic.

Cleanroom software engineering has demonstrated the ability to produce software in which the user finds no defects. We have confirmed these results at HP. This paper reports our results and provides a description of the cleanroom process, especially those portions of the process that we used.

Cleanroom Software Engineering

Cleanroom software engineering ("cleanroom") is a metaphor that comes from integrated circuit manufacturing. Large-scale integrated circuits must be manufactured in an environment that is free from dust, flecks of skin, and amoebas, among other things. The processes and environment are carefully controlled and the results are constantly monitored. When defects occur, they are considered to be defects in the processes and not in the product. These defects are characterized to determine the process failure that produced them. The processes are then corrected and rerun. The product is regenerated. The original defective product is not fixed, but discarded.

The cleanroom software engineering philosophy is analogous to the integrated circuit manufacturing cleanroom process. Processes and environments are carefully controlled and are monitored for defects. Any defects found are considered to be defects in one or more of the processes. For example, defects could be in the specification process, the design methodology, or the inspection techniques used. Defects are not considered to be in the source file or the code module that was generated. Each defect is characterized to determine which process failed and how the failure can be prevented. The failing process is corrected and rerun. The original product is discarded. This is why one of the main proponents of cleanroom, Dr. Harlan Mills, suggests that the most important tool for cleanroom is the wastebasket.[4]

Life Cycle

The life cycle of a cleanroom project differs from the traditional life cycle. The traditional 40-20-40 postinvestigation life cycle consists of 40% design, 20% code, and 40% unit testing. The product then goes to integration testing.

Cleanroom uses an 80-20 life cycle (80% for design and 20% for coding). The unexecuted and untested product is then presented to integration testing and is expected to work. If it doesn't work, the defects are examined to determine how the process should be improved. The defective product is then discarded and regenerated using the improved process.

No Unit Testing

Unit testing does not exist in cleanroom. Unit testing is private testing performed by the programmer with the results remaining private to the programmer.

The lack of unit testing in cleanroom is usually met with skepticism or with the notion that something wasn't stated correctly or it was misunderstood. It seems inconceivable that unit testing should not occur. However it is a reality. Cleanroom not only claims that there is no need for unit testing, it also states that unit testing is dangerous. Unit testing tends to uncover superficial defects that are easily found during testing, and it injects "deep" defects that are difficult to expose with any other kind of testing.

A better process is to discover all defects in a public arena such as in integration testing. (Preferably, the original programmer should not be involved in performing the testing.) The same rigorous, disciplined processes would then be applied to the correction of the defects as were applied to the original design and coding of the product.

In practice, defects are almost always encountered in integration testing. That seems to surprise no one. With cleanroom, however, these defects are usually minor and can be fixed with nothing more than an examination of the symptoms and a quick informal check of the code. It is very seldom that sophisticated debuggers are required.

When to Discard the Product

When IBM was asked about the criteria for judging a module worthy of being discarded, they stated that the basic criterion is that if testing reveals more than five defects per thousand lines of code, the module is discarded. This is a low defect density by industry standards,[5] particularly when it is considered that the code in question has never been executed even by an individual programmer. Our experience is that any half-serious attempt to implement cleanroom will easily achieve this. We achieved a defect density of one defect per thousand lines of code the first time we did a cleanroom project. It would appear that this "discard the offending module" policy is primarily intended to be a strong attention getter to achieve commitment to the process. It is seldom necessary to invoke it.

Productivity Is Not Degraded

Productivity is high with cleanroom. A trained cleanroom team can achieve a productivity rate approaching 800 noncomment source statements (NCSS) per engineer month. Industry average is 120 NCSS per engineer month. Most HP entities quote figures higher than this, but seldom do these quotes exceed 800 NCSS.

There is also evidence that the resulting product is significantly more concise and compact than the industry average.[6]

This further enhances productivity. Not only is the product produced at a high statement-per-month rate, but the total number of statements is also smaller.

Needed Best Practices

Cleanroom is compatible with most industry-accepted best practices for software generation. It is not necessary to unlearn anything. Some of these best practices are required (such as a structured design methodology). Others such as software reuse are optional but compatible.

As mentioned above, cleanroom requires some sort of structured design methodology. It has been successfully employed using a number of different design approaches. Most recently however, the cleanroom originators are recommending a form of object-oriented design.[7]

All cleanroom deliverables must be subject to inspections, code walkthroughs, or some other form of rigorous peer review. It is not critical what form is applied. What is critical is that 100% of all deliverables be subjected to this peer-review process and that it be done in small quantities. For instance, it is recommended that no more than three to five pages of a code module be inspected at a single inspection.

Required New Features

In addition to the standard software engineering practices mentioned above, there are a number of cleanroom-specific processes that are required or are recommended. These practices include structured specifications, functional verification, structured data, and statistical testing. Structured specifications are applied to the project before design begins. This strongly affects the delivery schedule and the project management process. Functional verification is applied during design, coding, and inspection processes. Structured data is applied during the design process. Finally, statistical testing is the integration testing methodology of choice. Fig. 2 summarizes the cleanroom processes.

Structured Specifications

Structured specifications[8] is a term applied to the process used to divide a product specification into small pieces for implementation. It is not critical exactly how this division is accomplished as long as the results have the following characteristics:

- Each specification segment must be small enough so that it can be fully implemented by the development team within days or weeks rather than months or years.
- The result of implementing each segment must be a module that can be completely executed and tested on its own. This means that no segment can contain partially implemented features that must be avoided during testing to prevent program failure.
- The segments may not have mutual dependencies. For example, it is satisfactory for segment 4 to require the implementation of segment 3 to execute correctly. It is assumed that segment 3 will be implemented first and will exist to support the testing of segment 4. However, it is not satisfactory for segment 3 to require segment 4 to execute properly at the same time.

The structured specifications process is used by cleanroom to facilitate control of the process by allowing the development team to focus on small, easily conceptualized pieces. A secondary but very important effect is that productivity is increased. Increased productivity is a natural effect of the team's being focused. Each deliverable is small and the time to produce it is psychologically short. The delivery date is therefore always imminent and always seems to be within reach. Morale is generally high because real progress is visible and is achievable.

Structured specifications also offer a very definite project management advantage. They serve to achieve the frequently quoted maxim that when a project is 50% complete, 50% of its features should be 100% complete instead of 100% of its features being 50% complete. Proper management visibility and the ability to control delivery schedules depend upon this maxim's being true.

Structured specifications are very similar to incremental processes described in other methodologies but often the purposes and benefits sound quite different. For instance, in one case the structured specifications process is called *evolutionary delivery.*[9] The primary benefit claimed for evolutionary delivery is that it allows "real" customers to examine early releases and provide feedback so that the product will evolve into something that really satisfies customer needs. HP supports this approach and has classes to teach the evolutionary delivery process to software developers.

From the description just given it would appear that each evolutionary release is placed into the hands of real customers. This implies to many people that the entire release process is repeated on a frequent (monthly) basis. Since multiple releases and the support of multiple versions are considered headaches for product support, this scenario is frowned

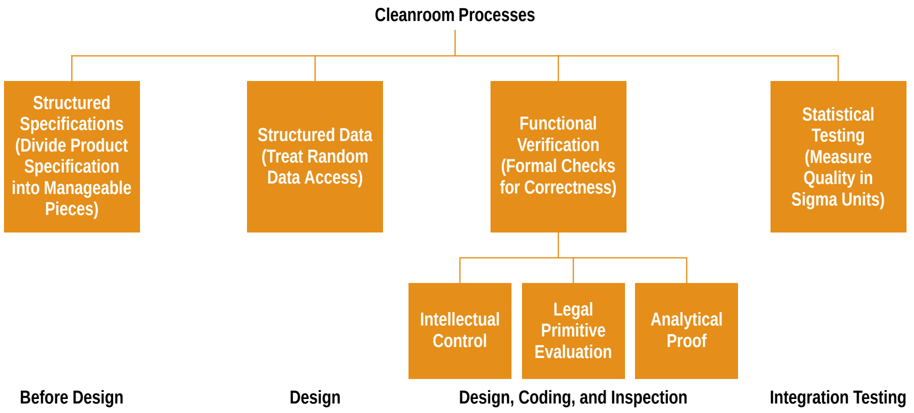

Cleanroom Processes
Structured Specifications (Divide Product Specification into Manageable Pieces)
Structured Data (Treat Random Data Access)
Functional Verification (Formal Checks for Correctness)
Statistical Testing (Measure Quality in Sigma Units)
Intellectual Control
Legal Primitive Evaluation
Analytical Proof
Before Design
Design
Design, Coding, and Inspection
Integration Testing

Fig. 2. The processes recommended for software cleanroom engineering.

upon. Cleanroom does not make it a priority to place each stage of the product into the hands of real customers.

Looking at the definition of a real customer in the evolutionary delivery process, you realize that a real customer could be the engineer at the next desk. In practice, the product cannot be delivered to more than a handful of alpha or beta testers until the product is released to the full market. This type of release should not occur any more often than normal. In fact, since cleanroom produces high-quality products, the number of releases required for product repair is significantly reduced.

Another type of structured specifications technique, which is applied to information technology development, uses information engineering time boxes.[10] Time boxes are used as a means of preventing endless feature creep while ensuring that the product (in this case an information product) still has flexibility and adaptability to changing business requirements.

HP has adopted a technique called *short interval scheduling*[11] as a project management approach. Short interval scheduling breaks the entire project into 4-to-6-week chunks, each with its own set of deliverables. Short interval scheduling can be applied to other projects besides those involved in software development. This is an insight that is not obvious in other techniques.

All of these methods are very similar to the structured specifications technique. As different as they sound, they all serve to break the task into bite-sized pieces, which is the goal of the structured specifications portion of cleanroom.

Functional Verification

Functional verification is the heart of cleanroom and is primarily responsible for achieving the dramatic improvement in quality possible with cleanroom. It is based on the tenet that, given the proper circumstances, the human intellect can determine whether or not a piece of logic is correct, and if it is not correct, devise a modification to fix it.* Functional verification has three levels: intellectual control, legal primitive evaluation, and analytical proof.

Intellectual control requires that the progression from specifications to code be done in steps that are small enough and documented well enough so that the correctness of each step is obvious. The working term here is "obvious." The reviewer should be tempted to say, "Of course this refinement level follows correctly from its predecessor! Why belabor the point?" If the reviewer is not tempted to say this, it may be advisable to redesign the refinement level or to document it more completely.

Legal primitive evaluation enhances intellectual control by providing a mathematically derived set of questions for proving and testing the assumptions made in the design specifications. Analytical proof[12] enhances legal primitive evaluation by answering the question sets mathematically. Analytical proof is a very rigorous and tedious correctness proof and is very rarely used.

We have demonstrated here at HP that intellectual control alone is capable of producing code with significantly improved defect densities compared to software developed with other traditional development processes. Application of the complete cleanroom process will provide another two to three orders of magnitude improvement in defect densities and will produce six-sigma code.

Intellectual Control. The human intellect, fallible though it may be, is able to assess correctness when presented with reasonable data in a reasonable format. *Testing is far inferior to the power of the human intellect.* This is the key point. All six-sigma software processes revolve around this point. It is a myth that software must contain defects. This myth is a self-fulfilling prophecy and prevents defect-free software from being routinely presented to the marketplace. The prevalence of the defect myth is the result of another myth, which is that the computer is superior to the human and that computer testing is the best way to ensure reliable software.

We are told that the human intellect can only understand complexities when they are linked together in close, simple relationships. This limitation can be made to work for us. If it is ignored, it works against us and handicaps our creative ability. Making this limitation work for us is the basis of functional verification.

The basis for intellectual control and functional verification is a structured development hierarchy. Most of us are familiar with a representation of a hierarchy like the one modeled in Fig. 3. This could be an illustration of how to progress from design specifications to actual code using any one of

* This tenet is also the definition of intellectual control.

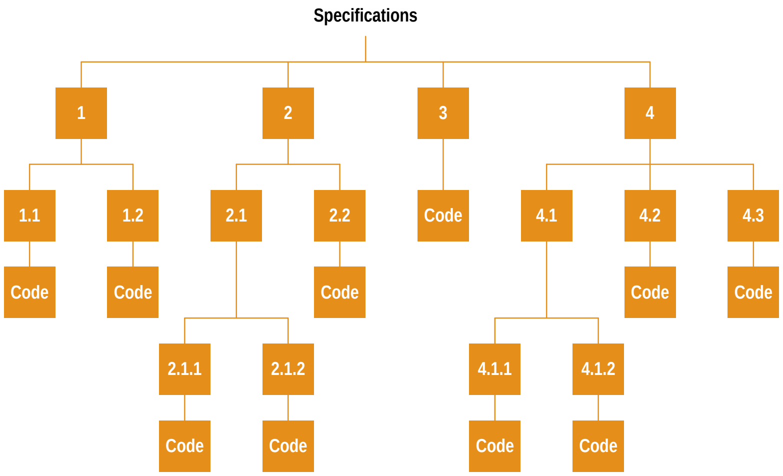


Fig. 3. A typical representation of a hierarchical diagram. In this case the representation is for a software design.

the currently popular, industry-accepted best practices for design. Each of these practices has some form of stepwise refinement. Each breaks down the specifications into ever greater detail. The result is a program containing a set of commands in some programming language.

The difference between the different software design methods is reflected in the interpretation of what the squares and the connecting lines in Fig. 3 represent. If the developer is using structured design techniques, they would mean data and control connections, and if the developer is using object-oriented design, they would represent objects in an object-oriented hierarchy.

Functional verification does not care what these symbols represent. In any of these methods, the squares 1, 2, 3, and 4 are supposed to describe fully the functionality of the specifications at that level. Similarly, 4.1 through 4.3 fully describe the functionality of square 4, and the code of square 3 fully implements the functionality of square 3. Intellectual control can be achieved with any of them by adhering to the following five principles.

Principle 1. Documentation must be complete. The first key principle is that the documentation of the refinement levels must be complete. It must fully reflect the requirements of the abstraction level immediately above it. For instance, it must be possible to locate within the documentation of squares 1 through 4 in our example every feature described in the specifications.

If documentation is complete, intellectual control is nearly automatic. In the case described above, the designer intuitively works to make the documentation and the specifications consistent with each other. The inspectors intuitively study to confirm the correctness.

Note that it is not always necessary to reproduce the specifications word for word. It will often be possible to simply state, “This module fully implements the provisions of specification section 7-4b.” The inspectors need only confirm that it makes sense for section 7-4b to be treated in a single module.

Other times it may be necessary to define considerably more than what is in the specifications. A feature that is spread over several modules requires a specific description of which portion is treated in each module and exactly how the modules interact with each other. It must be possible for the inspectors to look at all the modules as a whole and determine that the feature is properly implemented in the full module orchestration.

This principle is commonly violated. All industry-accepted best design processes encourage full documentation, but it is still not done because these design processes often lack the perspective and the respect for intellectual control that is provided by the principles of functional verification, or they are insufficiently compelling to convey this respect. The concept of intellectual control is often lost by many design processes because the main emphasis is on the mechanics of the specific methodology.

The result is that frequently the documentation for the first level of the system specifications is nothing more than the names of the modules (e.g., 1. Data Base Access Module, 2. In-Line Update Module, 3. Initialization Module, 4. User Interface Module). It is left to the inspectors to guess, based on the names, what portions of the specifications were intended to be in which module.

Even when there is an attempt to conform fully to the methodology and provide full documentation, neither the designer nor the inspectors seem to worry about the continuity that is required by functional verification. For example, a feature required in the design specification might show up first in level 4.1.2 or in the code associated with level 4.1.2 without ever having been referenced in levels 4 or 4.1. Sometimes the chosen design methodology does not sufficiently indicate that this is dangerous. Once again, this is the result of a failure to appreciate and respect the concept of intellectual control.

If proper documentation practices are followed, the result of each inspection is confidence that each level fully satisfies the requirements. For example, squares 1 through 4 in Fig. 3 fully implement the top-level specifications. Nothing is left out, deferred, undefined, or added, and no requirements are violated. Similarly, 4.1 through 4.3 fully satisfy the provisions of 4, and 4.1.1 and 4.1.2 fully satisfy 4.1.

With these conditions met, inspections of 4.1 through 4.3 should only require reference to the definition for square 4 to confirm that 4.1 through 4.3 satisfy 4. If 4.2 attempts to implement a feature of the specifications that is not explicitly or implicitly referenced in 4, it is a defect and should be logged as such in the inspection meeting.

Principle 2. A given definition and all of its next-level refinements must be covered in a single inspection session.

This means that a single inspection session must cover square 4 and all of its next-level refinements, 4.1 through 4.3. Altogether, 4.1 through 4.3 should not be more than about five pages of material. More than five pages would indicate that too much refinement was attempted at one time and intellectual control probably cannot be maintained. The offending level should be redone with some of the intended refinement deferred to a lower level.

Principle 3. The full life cycle of any data item must be totally defined at a single refinement level and must be covered in a single inspection.

This is the key principle that allows us to be able to inspect 2.1.1 and 2.1.2 and only be concerned about their reaction with each other and the way they implement 2.1. There is no need to determine, for example, if they interact correctly with 1.1 or 4.3.

This principle is a breakthrough concept and obliterates one of the most troublesome aspects of large-system modification. One seems never to be totally secure making a code modification. There’s always the concern that something may be getting broken somewhere else. This fear is an intuitive acknowledgment that intellectual control is not being maintained.

Such “remote breaking” can only occur because of inconsistent data management. Even troublesome problems associated with inappropriate interruptability or bogus recursion are caused by inconsistent data management. Intellectual control requires extreme respect for data management visibility.

This visibility can be maintained by ensuring that each data item is fully defined on a single abstraction level and totally studied in a single inspection session. It should be clear:

- Where and why the data item comes into existence
- What each data item is initialized to and why
- How and where each data item is used and what effects occur as a result of its use
- How and where each data item is updated and to what value
- Where, why, and how each data item is deleted.

Note that careful adherence to this principle contributes significantly to creating an object-oriented result even if that is not the intent of the designer. This principle is also one of the reasons why cleanroom lends itself so well to object-oriented design methodologies.

Once the inspection team is fully satisfied that the data management is consistent and correct, there is no need to be concerned about interactions. For instance, the life cycle for data that is global to the entire module would be fully described and inspected when squares 1, 2, 3, and 4 were inspected. Square 2 then totally defines its own portion of this management and 2.1, 2.2, 2.1.1, and 2.1.2 need only be concerned that they are properly implementing square 2's part of this definition. Squares 1, 3, and 4 can take care of their own portion with no worry about the effects on 2.

Adherence to principle 3 means that it is not necessary to inspect any logic other than that which is presented in the inspection packet. There is no intellectually uncontrolled requirement to execute the entire program mentally to determine whether or not it works.

Principle 4. Updates must conform to the same mechanisms.

Since even the best possible design processes are fallible, it is likely that unanticipated requirements will later be discovered. Functional verification does not preclude this. For instance, it may be discovered that it is necessary to test a global flag in the code for 2.1.1 which in turn must be set in the code for 4.2. This is a common occurrence and the typical response is simply to create the global flag for 2.1.1 and then update 4.2 to set it properly. Bug found. Bug fixed. Everything works fine.

However, we have just destroyed the ability to make subsequent modifications to this mechanism in an intellectually controlled way. Future intellectual control requires that this new interface be retrofitted into the higher abstraction levels. The life cycle of this flag must be fully described at the square 1 through 4 level. In that one document, the square 2 test and the square 4 update must be described, and then the appropriate portions of this definition must be repeated and refined in 2.1, 2.1.1, and 4.2, and of course, all of this should be subject to a full inspection.

Principle 5. Intellectual control must be accompanied by bottom-up thinking.

These principles can lull people into believing that they have intellectual control when, in fact, intellectual control is not possible. Intellectual control is, by its nature, a top-down process and is endangered by a pitfall that threatens all top-down design processes: the tendency to postpone real decisions indefinitely. To avoid this pitfall, the designers must be alert to potential "and-then-a-miracle-happens" situations. Anything that looks suspiciously tricky should be prototyped as soon as possible. All the top-down design discipline in the world will not save a project that depends upon a feature that is beyond the current state of the art. Such a feature may not be recognized until very late in the development cycle if top-down design is allowed to blind the developers to its existence.

The Key Word Is "Obvious." It must be remembered that these five principles are followed for the single purpose of making it obvious to the moderately thorough observer that the design is correct. Practicality must be sufficiently demonstrated, documentation must be sufficiently complete, the design must be tackled in sufficiently small chunks, and data management must be sufficiently clarified. All of these must be so obviously sufficient that the reviewer is tempted to say, "Of course! It's only obvious! Why belabor it?" If this is not the case, a redesign is indicated.

Our experience suggests that the achievement of such a state of obviousness is not a particularly challenging task. It requires care, but, if these principles are well understood, this care is almost automatic.

Legal Primitive Evaluation

Legal primitive evaluation enhances intellectual control by providing a mathematically derived set of questions for each legal Dykstra primitive (e.g., If-Then-Else, While-Do, etc.). For each primitive, the designers and the reviewers ask the set of questions that apply to that primitive and confirm that each question can be answered affirmatively. If this is the case, the correctness of the primitive is ensured.

A rigorous derivation of these questions can be found elsewhere.[13] There is insufficient space here to go through these derivations in detail, but we can illustrate the process and its mathematical basis by using a short, nonrigorous analysis of one of these sets, the While-Do primitive. Questions associated with the other primitives are given on page 47.

The While-Do construct is defined as follows:

S = [While A Do B;]

which means:

S is fully achieved by [While A Do B;].

The symbol S denotes the specification that the primitive is attempting to satisfy, or the function it is attempting to perform. The symbol A is the while test, and B is the while body.

As an example, S could be the specification: "The entry is added to the table." The predicate represented by A would then be an appropriate process to enable the program to perform an iteration and to determine if the operation is complete. B would be the processing required to accomplish the addition to the table. We have chosen to use a While-Do because, presumably, we think it makes sense. We may be intending to accomplish the entry addition by scanning the table sequentially until an appropriate insertion point is found and then splicing the entry into the table at that point. Whether or not this makes sense depends upon the known characteristics of the entry and the table. It also may depend upon the explicit (or implicit) existence of a further part of the specification such as "…within 5 ms."

To investigate whether it has been coded correctly, the following three questions are asked:

1. Is loop termination guaranteed for any argument of S?
2. When A is true, does S equal B followed by S?
3. When A is false, does S equal S?

When the answer to these three questions is yes, the correctness of the While-Do is guaranteed. The people asking these questions should be the designer and the inspectors.

These questions require some explanation.

1. Is loop termination guaranteed for any argument of S?

This means that for any data presented to the function defined by S, will the While-Do always terminate? For instance, in our example, are there any possible instances of the entry or the table for which the While-Do will go into an endless loop because A can never acquire a value of FALSE?

This would appear to be an obvious question. So obvious, that the reader may be tempted to ask why it is even mentioned. However, there is a lack of respect for While-Do termination conditions and many defects occur because of failure to terminate for certain inputs. A proper respect for this question will cause a programmer to take care when using it and will significantly help to avoid nontermination failures.

Respect for this question is justified because it is difficult to prove While-Do termination. In fact, it can be mathematically proven that, for the general case, it is impossible to prove termination.[14] To guarantee the correctness of a While-Do, it is therefore necessary to design simple termination conditions that can be easily verified by inspection. Complicated While-Do tests must be avoided.

2. When A is true, does S equal B followed by S?

This means that, when A is true, can S be achieved by executing B and then presenting the results to S again? This question is not quite so obvious.

Iterative statements are very difficult to prove. To prove the correctness of the while statement, it is desirable to change it to a noniterative form. We change it by invoking S recursively. Thus, the expression:

$$S = [\text{While A Do B};] \tag{1}$$

becomes:

$$S = [\text{If A Then (B; S)};] \tag{2}$$

Expression 2 is no longer an iterative construct and can be more readily proven. Fig. 4 shows the diagrams for these two expressions.

The equivalence of these two statements can be rigorously demonstrated.[15] A nonrigorous feeling for it can be obtained by observing that when A is true in [While A Do B], the B expression is executed once and then you start at the beginning by making the [While A] test again. If [While A Do B] is truly equal to S, then one could imagine that, rather than starting again at the beginning with the [While A] test, you simply start at the beginning of S. That changes the While-Do to a simple If-Then, and the predicate A is tested only once. If it is true, you execute B one time and then execute S to finish the processing.

The typical first reaction to this concept is that we haven't helped at all. The S expression is still iterative and now we've made it recursive making it seem that we have more to prove. The response to this complaint is that we don't have to prove anything about S at all. The specification (the entry is added to the table) is neither iterative nor recursive. We have simply chosen to implement it using a While-Do construct. We could, presumably, have implemented it some other way.

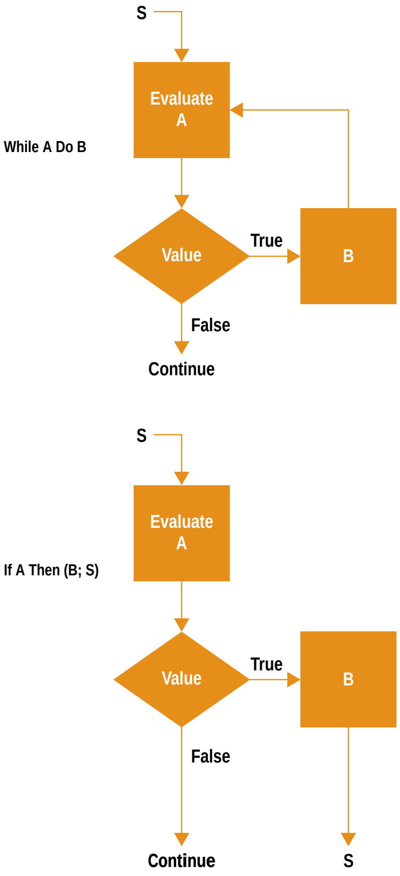


Fig. 4. Diagrams of the primitives While A Do B and If A Then (B; S).

S is nothing more than the specification. In the general case, it may be a completely arbitrary statement from any source. Whether the specification is correct or not is not our responsibility. Our responsibility is to implement it as defined.

Question 2 can therefore be restated as follows: If A is true, when we execute B one time and then turn the result over to whatever we've defined S to be, does the result still achieve S? An affirmative answer satisfies question 2.

In terms of our example, B will have examined part of the table. It will either already have inserted the new entry into the table or it will have decided that the portion of the table it examined is not a candidate for inserting of the entry. The unexamined portion of the table is now the new table upon which the construct must execute. This new instance of the table must be comparable to a standalone instance of the table so that the concept of adding an entry to the table still makes sense. If the resulting table fragment no longer looks like any form of the table for which the specification S was generated, question 2 may not be answerable affirmatively and the proposed code would then be incorrect.

3. When A is false, does S equal S?

This question seems fairly obvious but it is frequently overlooked. If A is found to be false the first time the While-Do is executed and therefore no processing of B occurs, is this satisfactory? Does the specification S allow for nothing to happen and therefore for no change to occur as a result of its execution?

In our example, the test posed by this question would likely fail. S requires something to happen (i.e., an entry to be added to the table). This would suggest that the While-Do may not be the appropriate construct for this S. We may never have noticed this fact if we hadn't been forced to examine question 3 carefully.

Structured Data

The principle of structured data[16] recognizes that undisciplined accesses to randomly accessed arrays or accesses that use generalized pointers cause the same kind of "reasoning explosion" produced by the undisciplined use of GOTOs. For instance, take the instruction:

```
A[i] = B[j+k];
```

This statement looks innocent enough. It would appear to be appropriate in any well-structured program. Note, however, that it involves five variables, all of which must be accounted for in any correctness analysis. If the program in which this statement occurs is such that this statement is executed several times, some of these variables may be set in instructions that occur later in the program. Thus, this instruction all by itself creates a reasoning explosion.

Just as Dykstra suggested that GOTOs should not be used at all,[17] the originators of cleanroom suggest that randomly accessed arrays and pointers should not be used. Dykstra recommended a set of primitives to use in place of GOTOs.

In the same way, cleanroom recommends that randomly accessed arrays be replaced with structures such as queues, stacks, and sets. These structures are safer because their access methods are more constrained and disciplined. Many current object-oriented class libraries support these structures directly and take much of the mystery and the complexity out of mentally converting from random-array thinking.

Statistical Testing

Statistical testing[8] is not really required for cleanroom, but it is highly recommended because it allows an assessment of quality in sigma units. It does not measure quality in defects per lines of code. Measuring quality in sigma units gives users visibility of how often a defect is expected to be encountered. For instance, it makes no difference if there are 100 defects per thousand lines of code if the user never actually encounters any of them. The product would be perceived as very reliable. On the other hand, the product may have only one defect in 100,000 lines of code, but if the user encounters this defect every other day, the product is perceived to be very unreliable.

Statistical testing also clearly shows when testing is complete and when the product can safely be released. If the model is predicting that the user will encounter a defect no more often than once every 5000 years with an uncertainty of ±1000 years, it could be decided that it is safe to release the product. This is usually better than some industry-standard methods (e.g., when the attrition rate from boredom among the test team exceeds a certain threshold, it must be time to release, or "When is this product supposed to be released? May 17th. What's today's date? May 17th. Oh. Then we must be finished testing.").

Legal Primitive Evaluation

As described in the accompanying article, the process of doing legal primitive evaluation involves asking a set of mathematically derived questions about the of basic programming primitives (e.g., If-Then-Else, For-Do, etc.) used in a program. The following is a list of the questions that must be investigated for each primitive.

In the following list S refers to the specification that must be satisfied by the questions asked about the referenced primitive.

- Sequence S = [A; B;]
  Does S equal A followed by B?
- For-Do S = [For A Do B;]
  ○ Does S equal first B followed by second B … followed by last B?
- If-Then S = [If A Then B;]
  ○ If A is true, does S = B?
  ○ If A is false, does S = S?
- If-Then-Else S = [If A Then B Else C;]
  ○ If A is true, does S equal B?
  ○ If A is false, does S equal C?
- Case S = [Case P part (C1) B1 … part (Cn) Bn Else E;]
  ○ When p is C1, does S equal B1?
  .
  .
  .
  When p is Cn, does S equal Bn?
  When p is not a member of set (C1, …, Cn), does S equal E?
- While-Do S = [While A Do B;]
  ○ Is loop termination guaranteed for any argument of S?
  ○ When A is true, does S equal B followed by S?
  ○ When A is false, does S equal S?
- Do-Until S = [Do A Until B;]
  ○ Is loop termination guaranteed for any argument of S?
  ○ When B is false, does S equal A followed by S?
  ○ When B is true, does S equal A?
- Do-While-Do S = [Do$_1$ A While B Do$_2$ C;]
  ○ Is loop termination guaranteed for any argument of S?
  ○ When B is true, does S equal A followed by C followed by S?
  ○ When B is false, does S equal A?

Statistical testing specifies the way test scenarios are developed and executed. Testing is done using scenarios that conform to the expected user profile. A user profile is generated by identifying all states the system can be in (e.g., all screens that could be displayed by the system) and, on each one, identifying all the different actions the user could take and the relative percentage of instances in which each would be taken. As the scenario generator progresses through these states, actions are selected randomly with a weighting that corresponds to the predicted user profile.

For instance, if a given screen has a menu item that is anticipated to be invoked 75% of the time when the user is in that screen, the invocation of this menu item is stipulated in 75% of the generated scenarios involving the screen. If another menu item will only be invoked 1% of the time, it would be called in only 1% of the scenarios.

These scenarios are then executed and the error history is evaluated according to a mathematical model designed to predict how many more defects the user would encounter in a given period of time or in a given number of uses. There are several different models described in the literature.[18]

In general, statistical testing takes less time than traditional testing. As soon as the model predicts a quality level corresponding to a predefined goal (e.g., six sigma) with a sufficiently small range of uncertainty (also predefined), the product can be safely released. This is the case even when 100% testing coverage is not done, or when 100% of the pathways are not executed.

Statistical testing requires that the software to which it is applied be minimally reliable. If an attempt is made to apply it to software that has an industry-typical defect density, any of the statistical models will demonstrate instabilities and usually blow up. When they don't blow up, their predictions are so unfavorable that a decision is usually made to ignore them. This is an analytical reflection of the fact that you can't test quality into a program.

Quality Cannot Be Tested into a Product

Although it is the quality strategy chosen for many products, it is not possible to test quality into a product. DeMarco[19] has an excellent analysis that demonstrates the validity of this premise. This analysis is based on the apparent fact that only about half of all defects can be eliminated by testing, but that this factor of two is swamped by the variability of the software packages on the market. The difference in defect density between the best and worst products is a factor of almost 4000.* Of course, these are the extremes. The factor difference between the 25th percentile and the 75th percentile is about 30 according to DeMarco. No one suggests that testing should not be done—it eliminates extremely noxious defects which are easy to test for—but compared to the variability of software packages, the factor of two is almost irrelevant. What then are the factors that produce quality software?

Capers Jones[20] suggests that inspections alone can produce a 60% elimination of defects, and when testing is added, 85% of defects are eliminated. There is no reported study, but the literature would suggest that inspections coupled with functional verification would eliminate more than 90% of defects.[21] Remarkably enough, testing seems to eliminate most (virtually all) of the remaining defects. The literature typically reports that no further defects are found after the original test cycle is complete and that none are found in the field.[21] This was also our experience.

There is apparently a synergism between functional verification and testing. Functional verification eliminates defects that are difficult to detect with testing. The defects that are left after application of inspections and functional verification are generally those that are easy to test for. The result is that >99% of all defects are eliminated via the combination of inspections, functional verification, and testing. Table II summarizes the percentage of defect removal with the application of individual or combinations of different defect detection strategies.

Table II
Defect Removal Percentages
Based on Defect Detection Strategies

Detection Strategy	% Defect Removal
Testing	50%
Inspections	60%
Inspections + Testing	85%
Inspections + Functional Verification	90%
Inspections + Functional Verification + Testing	>99%

Our Experience

We applied cleanroom to three projects, although only one of them actually made it to the marketplace. The project that made it to market had cleanroom applied all the way through its life cycle. The other projects were canceled for nontechnical reasons, but cleanroom was applied as long as they existed. The completed project, which consisted of a relatively small amount of code (3.5 KNCSS), was released as part of a large Microsoft® Windows system. The project team for this effort consisted of five software engineers.

All the techniques described in this paper except structured data and statistical testing were applied to the projects. All the products were Microsoft Windows applications written in C or C++. Structured data was not addressed because we never came across a serious need for random arrays or pointers. Although statistical testing was not applied, it was our intent eventually to do so, but the total lack of defects demotivated us from pursuing a complicated, analytical testing mode particularly when our testing resources were in high demand from the organization to help other portions of the system prepare for product release.

Design Methodology. We applied the rigorous object-oriented methodology known as box notation.[7] This is the methodology recommended by the cleanroom originators. We found it to be satisfyingly rigorous and disciplined.

Box notation is a methodology that progresses from functional specification to detailed design through a series of steps represented as boxes with varying transparency. The first box is a black box signifying that all external aspects of the system or module are known but none of the internal implementation is known. This is the ultimate object. It is defined by noting all the stimuli applied to the box by the user and the observable responses produced by these stimuli.

Inevitably, these responses are a function not only of the stimulus, but also of the stimulus history. For example, a mouse click at location 100,200 on the screen will produce a response that depends upon the behavior of the window that currently includes the location 100,200. The window at that location is, in turn, a function of all the previous mouse clicks and other inputs to the system.

* This factor is based on a defect density of 60 defects per KNCSS for the worst products and 0.016 defects per KNCSS for the best products. The factor difference between these two extremes is 60/0.016 = 3750 or ~4000.

The black box is then converted to a state box in which the stimulus history producing the responses of the black box is captured in the form of states that the box passes through. The response produced by a given stimulus can be determined not necessarily from the analysis of a potentially infinite stimulus history, but more simply by noting the state the system is in and the response produced by that stimulus within that state. States are captured as values within a set of state data. The state box fully reveals this data. It contains an internal black box that takes as its input the stimulus and the current set of state data and produces the desired response and a new set of state data. The state data is fully revealed but the internal black box still hides its own internal processing.

The state box is then converted to a clear box in which all processing is visible. However, this processing is represented as a series of interacting black boxes in which the interactions and the relations are clearly visible but, once again, the black boxes hide their own internal processing. This clear box is the final implementation of the object. In this object, the encapsulated data and the methods to process it are clearly visible.

Each of these internal black boxes is then treated similarly in a stepwise refinement process that ends only when all the internal black boxes can be expressed as single commands of the destination language.

This process allows many of the pitfalls of object-oriented design and programming to be avoided by carefully illuminating them at the proper time. For instance, the optimum data encapsulation level is more easily determined because the designer is forced to consider it at a level where perspective is the clearest. Data encapsulation at too high a level degrades modularity and defeats "object orientedness," but data encapsulation at too low a level produces redundancies and multiple copies of the same data with the associated possibility of update error and loss of integrity. These pitfalls are more easily avoided because the designer is forced to think about the question at exactly that point in the design when the view of the system is optimum for such a consideration.

Inspections. We employed a slightly adapted version of the HP-recommended inspection method taught by Tom Gilb.[9] We found this method very satisfactory. Our minimal adaptation was to allow slightly more discussion during the logging meeting than Gilb recommends. We felt that this was needed to accommodate functional verification.

Functional Verification. No attempt was made to implement anything but the first level of functional verification—intellectual control. This was found to be easily implemented, and when the principles were adequately adhered to, was almost automatic. Inspectors who knew nothing about functional verification or intellectual control automatically accomplished it when given material that conformed to its principles and, amusingly, they also automatically complained when slight deviations from these principles occurred.

Structured Specifications. The project team called cleanroom's structured specifications process evolutionary delivery because of its similarity to the evolutionary delivery methodology mentioned earlier and because evolutionary delivery is more like our HP environment. Structured specifications were developed in a defense-industry environment where dynamic specifications are frowned upon and where adaptability is not a virtue. However, evolutionary delivery assumes a dynamic environment and encourages adaptability. Regardless of the differences, both philosophies are similar.

At first, both marketing and management were skeptical. They were not reassured by the idea that a large amount of time would elapse before the product would take shape because of the large up-front design investment and because some features would not be addressed at all until very late in the development cycle. They were told not to expect an early prototype within the first few days that would demonstrate the major features.

Very quickly, these doubts were dispelled. Marketing was brought into the effort during the early rigorous design stages to provide guidance and direction. They participated in the specification structuring and set priorities and desired schedules for the releases. They caught on to the idea of getting the "juiciest parts" first and found that they were getting real code very quickly and could have this real code reviewed by real users while there was still time to allow the users' feedback to influence design decisions. They also became enthusiastic about participating in the inspections during the top-level definitions.

Management realized that the evolutionary staged releases were coming regularly enough and quickly enough that they could predict very early in the development cycle which stage had a high possibility of being finished in time to hit the optimum release window. They could then adjust scope and priority to ensure that the release date could be reliably achieved.

Morale. The cleanroom literature claims that cleanroom teams have a very high morale and satisfaction level. This is attributed to the fact that they have finally been given the tools necessary to achieve the kind of quality job that everyone wants to do. Our own experience was that this occurred surprisingly quickly. People with remarkably disparate, scarcely compatible personalities not only worked well together, they became enthusiastic about the process.

It appears that the following factors were influential in producing high morale:

- Almost daily inspections created an environment in which each person on the team took turns being in the "hot seat." People quickly developed an understanding that reasonable criticism was both acceptable and beneficial. The resulting frankness and openness were perceived by all to be remarkably refreshing and exhilarating.
- Team members were surprised that they were being allowed to do what they were doing. They were allowed to take the time necessary to do the kind of job they felt was proper.

Productivity. Productivity was difficult to measure. Only one project actually made it to the market place, and it is difficult to divide the instruction count accurately among the engineers that contributed to it. However, the subjective impression was that it certainly didn't take any longer. When no defects are found one suddenly discovers that the job is finished. At first this is disconcerting and anticlimactic, but it also emphasizes the savings that can be realized at the end of the project. This compensates for the extra effort at the beginning of the project.

Conclusion

The cleanroom team mentioned in this paper no longer exists as a single organization. However, portions of cleanroom are still being practiced in certain organizations within Hewlett-Packard. These portions especially include structured specifications and intellectual control.

We believe our efforts can be duplicated in any software organization. There was nothing unique about our situation. We achieved remarkable results with less than total dogmatic dedication to the methodology.

The product that made it to market was designed using functional decomposition. Even though functional decomposition is minimally rigorous and disciplined, we found the results completely satisfactory. The project consisted of enhancing a 2-KNCSS module to 3.5 KNCSS.

The original module was reverse engineered to generate the functional decomposition document that became the basis for the design. The completed module was subjected to the intellectual control processes and the reviewers were never told which code was the original and which was modified or new code. A total of 36 defects were found during the inspection process for a total of 10 defects per KNCSS. An additional five defects were found the first week of testing (1.4 defects per KNCSS). No defects were encountered in the subsequent 10 months of full system integration testing and none have been found since the system was released.

It was interesting to note that the defects found during inspections included items such as a design problem which would have, under rare conditions, mixed incompatible file versions in the same object, a piece of data that if it had been accessed would have produced a rare, nonrepeatable crash, and a number of cases in which resources were not being released which would, after a long period of time, have caused the Windows system to halt. Most of these defects would have been very difficult to find by testing.

Defects found during testing were primarily simple screen appearance problems which were readily visible and easily characterized and eliminated. These results conform well to expected cleanroom results. About 90% of the defects were eliminated by inspections with functional verification. About 10% more were eliminated via testing. No other defects were ever encountered in subsequent full-system integration testing or by customers in the field. It can be expected on the basis of other cleanroom results reported in the literature that at least 99% of all defects in this module were eliminated in this way and that the final product probably contains no more than 0.1 defect per KNCSS.

References

1. M.J. Harry, *The Nature of Six Sigma Quality,* Motorola Government Electronics Group, 1987.

2. P.A. Tobias, "A Six Sigma Program Implementation," *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference,* p. 29.1.1.

3. H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software,* Vol. 4, no. 5, September 1987, pp. 19-25.

4. H.D. Mills and J.H. Poore, "Bringing Software Under Statistical Quality Control," *Quality Progress,* Vol. 21, no. 11, November 1988, pp. 52-55.

5. T. DeMarco, *Controlling Software Projects,* Yourdon Press, 1982, pp. 195-267.

6. R.C. Linger and H.D. Mills, "A Case Study in Software Engineering," *Proceedings COMPSAC 88,* p. 14.

7. H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design,* Academic Press, 1986.

8. P.A. Currit, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering,* Vol. SE-12, no. 1, January 1986, pp 3-11.

9. T. Gilb, *The Principles of Software Engineering Management,* Addison-Wesley, 1988, pp. 83-114.

10. J. Martin, *Information Engineering Book III,* Prentice Hall, 1990, pp. 169-196.

11. *Navigator Systems Series Overview Monograph,* Ernst & Young International, Ltd., 1991, pp. 55-56.

12. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley, 1979, pp. 227-229.

13. *Ibid,* pp. 213-300.

14. H.D. Mills, et al, *Principles of Computer Programming,* Allyn and Bacon, Inc., 1987, pp. 236-238.

15. R.C. Linger, H.D. Mills, and B.I. Witt, *op cit,* pp. 219-221.

16. H.D. Mills and R.C. Linger, "Data Structured Programming: Program Design without Arrays and Pointers," *IEEE Transactions on Software Engineering,* Vol. SE-12, no. 2, Feb. 1986, pp. 192-197.

17. E.W. Dijkstra, "Structured Programming," *Software Engineering Techniques,* NATO Science Committee, 1969, pp. 88-93.

18. P.A. Currit, M. Dyer, and H.D. Mills, *op cit,* pp. 3-11.

19. T. DeMarco, *op cit,* p. 216.

20. Unpublished presentation given at the 1988 HP Software Engineering Productivity Conference.

21. R.C. Linger and H.D. Mills, *op cit,* pp. 8-16.

Fuzzy Family Setup Assignment and Machine Balancing

Fuzzy logic is applied to the world of printed circuit assembly manufacturing to aid in balancing machine loads to improve production rates.

by Jan Krucky

In disciplines such as engineering, chemistry, or physics, precise, logical mathematical models based on empirical data are used to make predictions about behavior. However, some aspects of the real world are too imprecise or "fuzzy" to lend themselves to modeling with exact mathematical models.

The tool we have for representing the inexact aspects of the real world is called fuzzy logic. With fuzzy logic, we can model the imprecise modes of reasoning that play a role in the human ability to make decisions when the environment is uncertain and imprecise. This ability depends on our aptitude at inferring an approximate answer to a question from a store of knowledge that is inexact, incomplete, and sometimes not completely reliable. For example, how do you know when you are "sufficiently close" to but not too far away from a curb when parallel parking a car?

In recent years fuzzy logic has been used in many applications ranging from simple household appliances to sophisticated applications such as subway systems. This article describes an experiment in which fuzzy logic concepts are applied in a printed circuit assembly manufacturing environment. Some background material on fuzzy logic is also provided to help understand the concepts applied here.

The Manufacturing Environment

In printed circuit assembly environments, manufacturers using surface mount technology are concerned with machine setup and placement times. In low-product-mix production environments manufacturers are primarily concerned with placement time and to a lesser degree setup time. In medium-to-high-product-mix production environments manufacturers are mainly concerned with setup time.

One solution to the setup problem is to arrange the printed circuit assemblies into groups or families so that the assembly machines can use the same setup for different products. In other words, reduce or eliminate setups between different assembly runs. The solution to minimizing placement time is to balance the component placement across the placement machines.

HP's Colorado Computer Manufacturing Operation (CCMO) is a medium-to-high-product-mix printed circuit assembly manufacturing entity. The heuristic, fuzzy-logic-based algorithms described in this paper help determine how to minimize setup time by clustering printed circuit assemblies into families of products that share the same setup and by balancing a product's placement time between multiple high-speed placement process steps.

The Placement Machines

The heart of our surface mount technology manufacturing lines in terms of automated placement consists of two Fuji CP-III high-speed pick-and-place machines arranged in series and one Fuji IP-II general-purpose pick-and-place machine.

A Fuji CP-III placement machine supports two feeder banks each having 70 slots available for component feeders to be mounted on (see Fig. 1). The components are picked from their feeders and placed on the printed circuit board, creating a printed circuit assembly. A component feeder might take one or two slots. The tape-and-reel type feeder, which is the one we use at CCMO, is characterized by its width for slot allocation purposes. The standard feeder tape widths are 8 mm, 12 mm, 16 mm, 24 mm, and 32 mm. The 8-mm feeder tapes consume one slot each while the 12-mm to 32-mm feeder tapes consume two slots. Additional feeder-to-feeder spacing constraints might increase the number of slots the feeders actually require. A component's presentation, package type, and style determine the tape-and-reel width and therefore the feeder size.

Split-Bank

A feature of the Fuji CP-III called *split-bank* addresses the problem of high setup costs by allowing one bank to be

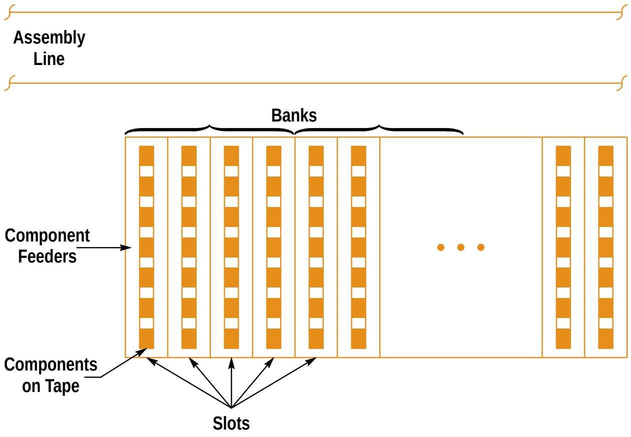


Fig. 1. Simplified representation of a tape-and-reel type placement machine.

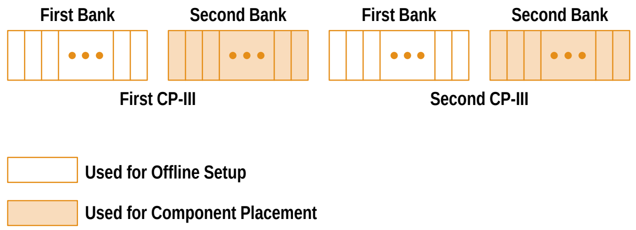


Fig. 2. The split bank feature of the Fuji CP-III assembly machine. The first feeder banks of each machine are used for offline setup.

used for component placement while the other bank is being set up offline. Fig. 2 illustrates this split-bank feature. In this configuration the first feeder bank on each machine is used to perform the offline setup, and the second bank is used for component placement.

Setup Time versus Placement Time

Our printed circuit assembly products vary quite a bit in their setup-slot requirements. They range from eight slots on the low end to 260 slots on the high end. For an average product requiring 45 slots it takes 45 online minutes to set up the feeders for placement. The average placement time is 2.5 minutes per board. In a low-to-medium-volume printed circuit assembly shop such as CCMO, the average lot size is 20 products. Therefore, for an average run of 20 products, 47% of the time is spent on setup (leaving the placement machine idle) and 53% of the time is spent placing the components. This constitutes an unacceptable machine use and hence a low output from the manufacturing shop. It's not just the online setup time, but also the frequency of having to do these setups that affects productivity and quality. A fixed setup for an entire run would seem to be a solution to this problem. However, in a medium-to-high-product-mix, low-volume environment such as ours, fixed setups cannot be used. HP CCMO currently manufactures 120 products with 1300 unique components placeable by the Fuji CP-III equipment.

Another quick solution to this online setup time problem would be to place a certain percentage of CP-III placeable components at a different process step. For example, we could use the Fuji IP-II for this purpose. This is not a feasible solution because the Fuji IP-II has a placement speed that is four times slower than the CP-III. We use the Fuji IP-II primarily for placing large components.

Alternate Setup Methodology

The setup time requirements mentioned above suggest that we needed an alternative setup methodology to minimize online setup time. The approaches we had available included expansion on the split-bank option described above, clustering the printed circuit assembly products into families with identical setup while still allowing the split-bank setup feature to be fully used, and balancing the two series CP-III placement loads as much as possible. Balancing implies that the components would be distributed between the two placement machines so that both machines are kept reasonably busy most of the time.

Since most of our printed circuit assembly products are double-sided, meaning that components are placed on the top and bottom sides of the printed circuit board, independent balancing for each side of the printed circuit assembly was considered. However, family clustering as viewed by the layout process dictated that a double-sided printed circuit assembly should be treated as a sum of the requirements for both sides of the board. Thus, no machine setup change would be required when switching from side A to side B of the same product.

Why Families

While clustering products into families is a viable and an attractive solution, other possible solutions such as partially fixed setups augmented by families or scheduling optimization to minimize the setup changes in the build sequence, are also worth consideration.

One can imagine that none of the solutions mentioned above will provide the optimal answer to every online setup-time issue, but their reasonable combination might. The following reasons guided us into choosing family clustering as an initial step towards minimizing online setup costs.

- Intuitive (as opposed to algorithmic) family clustering on a small scale has been in place at our manufacturing facility for some time.
- It appears that families give reasonable flexibility in terms of the build schedule affecting the entire downstream process.
- Families can take advantage of the CP-III's split-bank feature.
- By altering the time window of a particular family's assembly duration (i.e., by shift, day, week, month, and so on), one can directly control a family's performance and effectiveness.

We chose a heuristic approach to minimizing setup time because an exhaustive search is O(n!), where n is the number of products. Our facility currently manufactures 120 products and expects to add 40 new ones in the near future, which would make an exhaustive search unrealistic.

Primary Family

As explained above, we wanted to use the family clustering approach to take advantage of the CP-III split-bank setup feature. One can quickly suggest a toggle scenario in which each feeder bank would alternate between the states of being set up offline and being used for placement. However, it would be difficult to synchronize the labor-intensive activities of perpetual offline setups in a practical implementation. Also, this option would require a sufficient volume in the toggled families to allow the completion of offline setups at the idle placement banks. Given these reasons, a strict toggle approach would probably not have worked to improve the overall setup time in our environment. Instead of toggling, we selected an approach in which certain feeder banks are permanently dedicated to a family, and the remaining banks toggle between offline setup and placement. This approach led to the primary family concept.

A primary family is one that will not be toggled and is therefore always present on the machine. Since the primary family is permanently set up it logically follows that each nonprimary family includes primary family components. The more primary family slots used by a bank containing a nonprimary family the better. The summation of two series CP-III's banks provides four setup banks available on a line. We elected to dedicate the first bank of each CP-III to the primary family, leaving us with two banks for nonprimary families (see Fig. 3).

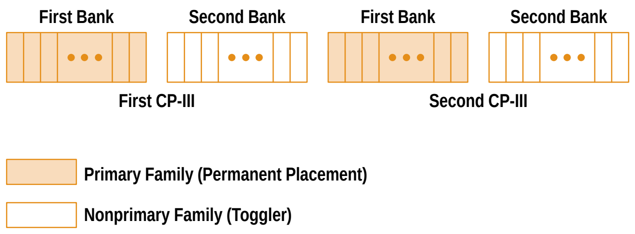


Fig. 3. The setup for primary and nonprimary families.

Fig. 4 shows how the primary family concept can be used to schedule a group of products to be built. Let's assume that we have primary family A and two nonprimary families AB and AC. (Products in families AB and AC contain components that are also part of products in family A.) First, any of family AB's products can be built. Although primary family A's products could be built using the same setup, it is highly undesirable since it would waste the presence of family AB's setup. So, after all the demand from family AB's products have been satisfied, we can switch to products in primary family A while we set up offline for family AC. On the practical side, it is unnecessary to set up the entire nonprimary family unless all the family's products are actually going to be built.

This example shows that the primary family concept is useful only if it is incorporated into the build schedule and the primary family must have sufficient product volume to allow the behavior depicted in Fig. 4.

Balancing

Family clustering results in a shared setup by a group of printed circuit assembly products which among other things might share the same components. This creates the problem of ensuring that the assembly of all products is adequately balanced on the two series CP-III placement machines. If the load is not balanced, an undesirable starvation in the process pipeline might occur. Balancing is accomplished by properly assigning the family's components between the two series CP-III machines. An intuitive guess suggests that the success of family clustering might make the balancing efforts proportionally harder. Although the online setup time reduction is the primary goal, it cannot justify a grossly imbalanced workload between two serial CP-IIIs. Therefore, machine balancing is the key that truly enables the family approach.

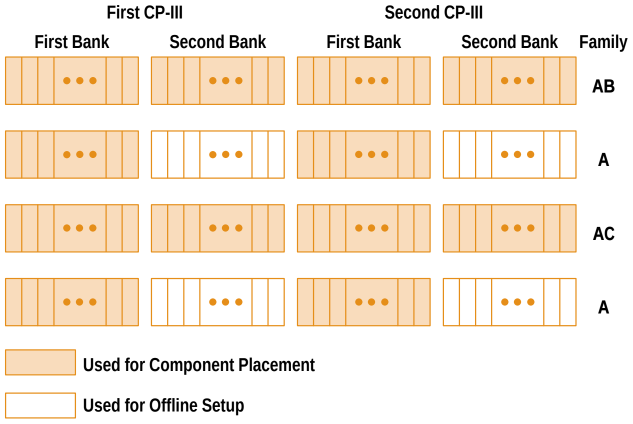


Fig. 4. An illustration of how the primary family concept can be used to schedule a group of products for assembly.

Other Methods

In addition to applying fuzzy set theory to solve our family assignment and balancing problem we also used two other approaches: the greedy board heuristic and an extension to the greedy board heuristic.

Greedy Board Approach. A research group at HP's strategic planning and modeling group in conjunction with Stanford University suggested the greedy board heuristic approach to minimize high setup cost for semiautomated manufacturing operations. HP's Networked Computer Manufacturing Operation (NCMO) implemented the greedy board approach at their site.[1]

In the greedy-board heuristic a family is defined by the repetitive addition of products, one at a time, until slot availability is exhausted. The selection criterion is a function of the product's expected volume and its additional slot requirements. The greedy ratio is:

$$G_i = s_i/v_i$$

where s_i is the number of additional slots a product p_i adds to the family, and v_i is the product's volume. Since the objective is to minimize the number of slots added while maximizing the family's volume, the product with the smallest greedy ratio wins and is added to the family. New slots are obtained and the selection process, via the greedy ratio, is repeated until either there are no more slots available or no more products are to be added. See the greedy board example on page 54.

The greedy board implementation at NCMO performs balancing by assigning components to the machines by a simple alternation until constraints are met. The components are initially sorted by their volume use. This approach balances the family overall, but it carries no guarantees for the products that draw from the family.

Extension to Greedy Board Heuristic. The greedy board heuristic tends to prefer smaller, high-volume printed circuit assemblies in its selection procedure. At CCMO we extended the original greedy ratio to:

$$G_i = \frac{s_i}{v_i \times c_i}$$

where c_i is an average number of slots product p_i shares with products not yet selected. The CCMO extension slightly curbs the volume greediness at the expense of including a simple measure of commonality. However, the results showed the CCMO extension to the greedy board heuristic performed slightly better than the original algorithm. The results from the two greedy-board approaches and the fuzzy approach are given later in this paper.

Despite the relatively good results achieved by our extension to the greedy board heuristic approach, we were still looking for an alternative approach. This led us to explore using fuzzy set theory to find a solution to our placement machine setup problem.

The Greedy Board Family Assignment Heuristic

As mentioned in the main article, a family in our manufacturing environment is a group of products (boards) that can be built with a single setup on the component placement machines. The greedy board heuristic is one way of assigning products to families for printed circuit assembly. The only data required for the greedy board algorithm is the list of components and the expected volume for each board. Each family is created by the repetitive addition of products, one at a time, until slot availability is exhausted.

In the following example assume there are eight component slots available per family and that the following boards must be assigned to families.

Board	Expected Volume (v_i)	Components
Alpha	1400	A, F, K, M
Tango	132	C, K
Delta	2668	H, D, R, F, K
Echo	1100	R, J, S, K
Beta	668	G, F, T, L
Lambda	1332	H, D, F, K
Gamma	900	A, J, E, K

The board with the lowest greedy ratio is the first one added to the current family being created.

Board	New Parts	Expected Volume	Greedy Ratio
	(s_i)	(v_i)	($G_i = s_i/v_i$)
Alpha	4	1400	0.0029
Tango	2	132	0.0152
Delta*	5	2668	0.0019
Echo	4	1100	0.0036
Beta	4	668	0.0060
Lambda	4	1332	0.0030
Gamma	4	900	0.0040

Delta is the board with the lowest greedy ratio so it becomes the first member of the family. It has the highest product volume added per component slot used. Delta adds five components, leaving three slots to fill this family.

With the components H, D, R, F, and K already in the family, for the next board the ratios are computed as follows:

Board	New Parts	Expected Volume	Greedy Ratio
	(s_i)	(v_i)	($G_i = s_i/v_i$)
Alpha	2	1400	0.0014
Tango	1	132	0.0076
Echo	2	1100	0.0018
Beta	3	668	0.0045
Lambda	0	1332	0.0000
Gamma	3	900	0.0033

The Lambda board is the one with the lowest greedy ratio because its components are a complete subset of the components already in the family. Since adding Lambda to the family does not require the addition of any components to the family, the greedy ratios given above still apply for the selection of the next board. The Alpha board has the next lowest ratio and it adds two new components (A and M) to the family. This brings the total number of components in the family to seven—one slot left.

After adding the Alpha board to the family, the new part-to-volume ratios for the remaining unassigned boards become:

Board	New Components	Expected Volume	Greedy Ratio
	(s_i)	(v_i)	($G_i = s_i/v_i$)
Tango	1	132	0.0076
Echo	2	1100	0.0018
Beta	3	668	0.0045
Gamma	3	900	0.0033

Now Echo has the lowest ratio. However, the Echo board has two components, and since we already have seven components, adding the Echo components to the family would exceed our limit of eight components per family. Therefore, Tango is the only board that will fit even though it has the lowest theoretical contribution. Adding the Tango board fills up the family allotment. Finally, the components in the family include H, C, D, A, R, F, K, and M.

The next family is defined by following the above procedure for the remaining boards: Echo, Beta, and Gamma.

Fuzzy Set Theory

The following sections provide a brief overview of some of the basic concepts of fuzzy set theory applicable to the topics discussed in this paper. For more about fuzzy set theory see reference 2.

Fuzzy Sets

Unlike the classical yes and no, or *crisp* (nonfuzzy) sets, fuzzy sets allow more varying or partial degrees of membership for their individual elements (see Fig. 5). Conceptually only a few natural phenomena could be assigned a crisp membership value of either yes or no without any doubt. On the other hand, most of the real-world's objects, events, linguistic expressions, or any abstract qualities we experience in our everyday life tend to be more suited for a fuzzier set membership. Fuzzy sets allow their elements to belong to multiple sets regardless of the relationship among the sets.

In spite of their tendency to seem imprecise, fuzzy sets are unambiguously defined along with their associated operations and properties. The fuzzy sets used in the fuzzy family assignment and machine balancing heuristic exist in universes of discourse that are finite and countable.

Definition. To begin our discussion of fuzzy sets we define the universe of discourse $X = \{x_1, x_2 \ldots x_i\}$ and let $\mu_A(x_i)$ denote the degree of membership for fuzzy set A on universe X for element x_i. The degree of membership function for fuzzy set A is $\mu_A(x) \in [0,1]$, where 0 represents the weakest membership in a set and 1 represents the strongest membership in a set.

$$\text{Fuzzy set A} = \frac{\mu_A(x_1)}{x_1} + \ldots + \frac{\mu_A(x_i)}{x_i}$$

where the horizontal bar is not a quotient but a delimiter.

Examples. The following examples show different types of fuzzy sets.

- Number as a fuzzy set:
  - Universe U = {0, 1, 2, 3, 4, 5}
  - Fuzzy set A = 0.2/0 + 0.7/1 + 0.8/2 + 0.2/3 + 0.1/4 + 0.0/5
  - Fuzzy set A might be described linguistically as "just about 2" because 0.8/2 has the highest degree of membership in fuzzy set A.
- Defining people in terms of their preference for certain alcoholic beverages:
  - Universe Y = {beer, wine, spirits} = $\{y_1, \ldots y_3\}$

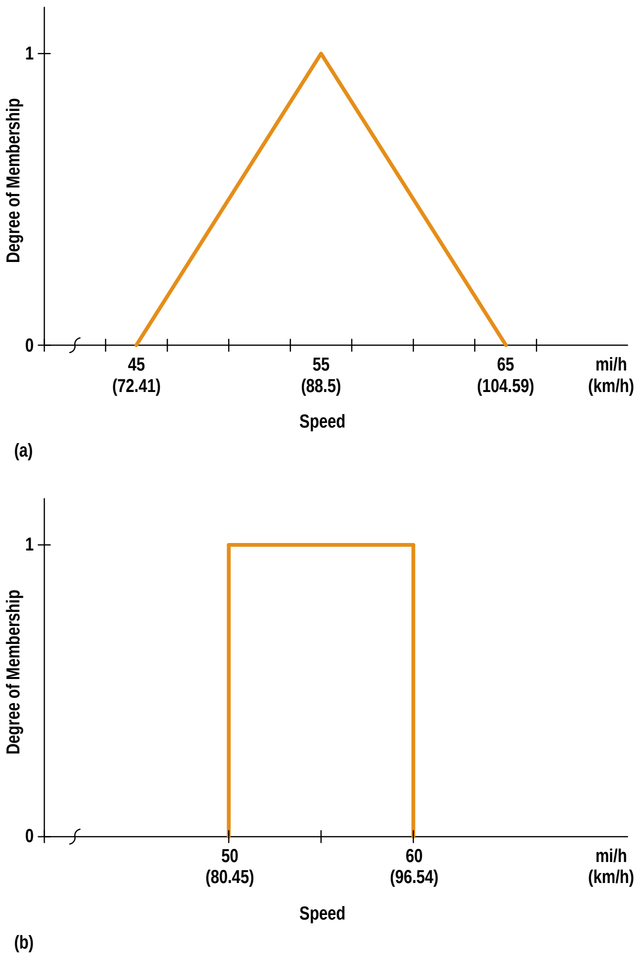


Fig. 5. Crispy and fuzzy representation of the notion of average driving speed. (a) In the fuzzy representation the membership class average driving speed varies from zero for <45 mi/h or >65 mi/h to 100% at 55 mi/h. (b) In a crisp representation membership in the average driving speed set is 100% in the range from 50 to 60 mi/h only.

- ○ Fuzzy set B = $\frac{\mu_B(y_i)}{y_i}$ = 0.1/beer + 0.3/wine + 0.0/spirits might describe somebody who doesn't drink much but prefers wine and dislikes spirits
- ○ Fuzzy set C = 0.8/beer + 0.2/wine + 0.1/spirits might describe a beer lover
- ○ Fuzzy set D = 0.0/beer + 0.0/wine + 0.0/spirits might describe a person who doesn't indulge in alcoholic beverages
- ○ Fuzzy set E = 0.8/beer + 0.7/wine + 0.8/spirits might describe a heavy drinker.

- Defining a person in terms of their cultural heritage:
  - ○ Universe Z = {Zirconia, Opalinia, Topazia} and $\mu_F(z_i)$ represents a degree of cultural heritage from the three provinces in some imaginary gem-producing country.
  - ○ Fuzzy set F = 0.3/Zirconian + 0.5/Opalinian + 0.1/Topazian might describe someone who was born in Western Opalina, attended a university in Zirconia, and married a Topazian living in Diamond City, Zirconia.
- Lunch hour:
  - ○ Universe W = Day (continuous time of 24 hours)
  - ○ The fuzzy set L (1100 to 1300) might represent the term "lunch hour" as shown in Fig. 6.

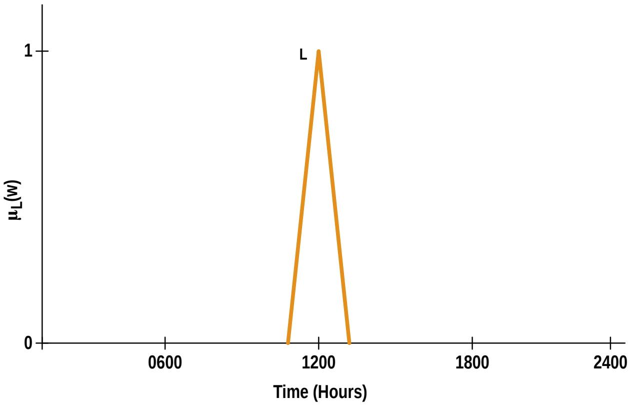


Fig. 6. A fuzzy set representation of the term lunch hour.

Operations. Operations such as union, intersection, and complement are defined in terms of their membership functions. For fuzzy sets A and B on Universe X we have the following calculations:

- Union: $\mu_{AUB}(x) = \mu_A(x) \vee \mu_B(x)$
  or $\forall x_i : \mu_{AUB}(x_i) = \mathrm{Max}(\mu_A(x_i), \mu_B(x_i))$
  (see Fig. 7a).

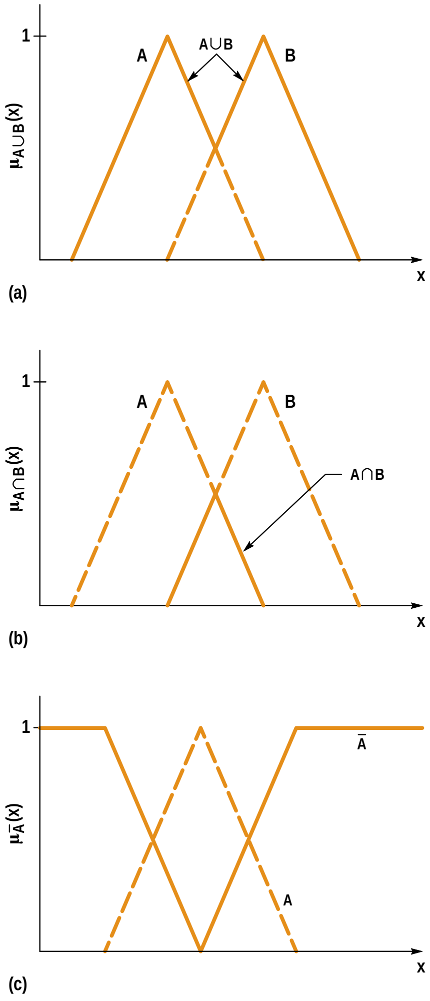


Fig. 7. Fuzzy set operations. (a) Union. (b) Intersection. (c) Complement.

- Intersection: $\mu_{A \cap B}(x) = \mu_A(x) \wedge \mu_B(x)$ or $\forall x_i : \mu_{A \cap B}(x_i) = \text{Min}(\mu_A(x_i), \mu_B(x_i))$ (see Fig. 7b).
- Complement: $\mu_{\bar{A}}(x) = 1 - \mu_A(x)$ or $\forall x_i : \mu_{\bar{A}}(x_i) = 1 - \mu_A(x_i)$ (see Fig. 7c).

All of the operations defined above hold for fuzzy or classical set theory. However, the two formulas known as excluded middle laws do not hold for fuzzy sets, that is:

$$A \cup \overline{A} = X$$
$$A \cap \overline{A} = 0$$

for classical set theory, but

$$A \cup \overline{A} \neq X$$
$$A \cap \overline{A} \neq 0$$

for fuzzy set theory.

These laws, which take advantage of the either-or only membership for a classical set's elements, cannot hold for fuzzy sets because of their varying degree of set membership. Fig. 8 provides a graphical comparison between these two formulas for classical and fuzzy set operations.

Fuzzification and Defuzzification

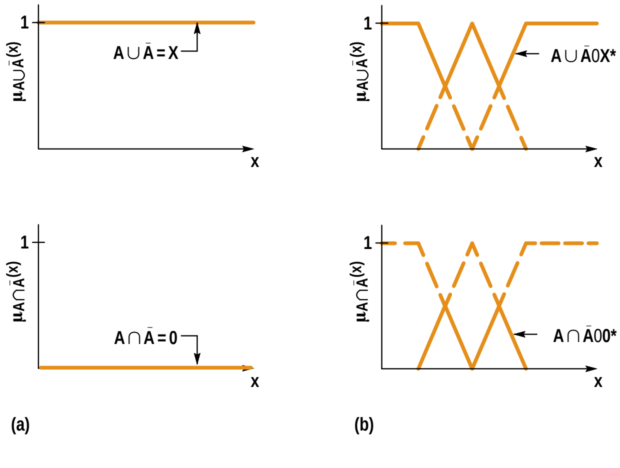
Fuzzification and defuzzification are operations that translate back and forth between fuzzy and crisp representations of information, measures, or events. Since most of our environment is more naturally represented in a fuzzy form rather than a crisp form, the need for a fuzzification step could be perceived as being a rare event. On the other hand, a defuzzification procedure is needed more often, as in the case in which a fuzzy set has to be expressed as a single crisp number. There are several defuzzification methods. One of the most commonly used and computationally trivial is the Max method. The Max method simply chooses an element with the largest membership value to be the single crisp representation of the fuzzy set. For example, for the fuzzy set C given above the Max defuzzification method would yield 0.8/beer (i.e., fuzzy set C describes a beer lover).

Fuzzy Relations

The concept of relations between fuzzy sets is fairly analogous to the idea of mapping in classical set theory in which the elements or subsets of one universe of discourse are mapped to elements or sets in another universe of discourse. For example, if A is a fuzzy set on universe X and B is a fuzzy set on universe Y then the fuzzy relation R = A ⊗ B maps universe X to universe Y (i.e., R is a relation on universe X **x** Y). The symbol ⊗ denotes a composition operation which computes the strength of the relation between the two sets. Please note that in general A ⊗ B ≠ B ⊗ A and furthermore A ≠ R ⊗ B.

Special Properties. The following are some of the special properties of fuzzy relations.

- A fuzzy set is also a fuzzy relation. For example, if A is a fuzzy set on universe X and there exists I = 1/y as an identity fuzzy set on Universe Y = {y}, then fuzzy relation R = A ⊗ I = A.
- The same operations and properties valid for fuzzy sets also hold for fuzzy relations.



*See Fig. 7c for Ā

Fig. 8. A comparison of the excluded middle laws for (a) classical sets and (b) fuzzy sets.

- Fuzzy logic implication of the form P → Q can be also represented by a fuzzy relation since T(P → Q) = T($\overline{P}$ ∨ Q) where T is the truth evaluation function. For example, if A is a fuzzy set on universe X and B is a fuzzy set on universe Y then a proposition P → Q describing IF A THEN B, is equivalent to the fuzzy relation R = (A ⊗ B) ∪ ($\overline{A}$ ⊗ Y). Fig. 9 shows a graphical representation of this relationship.

Fuzzy Composition

Fuzzy composition operations compute the strength of the relation between two fuzzy relations. To show the most popular composition operators, consider that we have fuzzy sets X, Y, and Z and that R is a fuzzy relation on universe X **x** Y and that S is a fuzzy relation on universe Y **x** Z. To find the fuzzy relation T = R ⊗ S on universe X **x** Z we use one of the following composition operations:

- Max-Min: $\mu_T(t_{i,j}) =$

$$\text{Max}\left[\text{Min}\left(\mu_R(r_{k,i}), \mu_S(s_{j,k})\right)\forall k : \leq \beta\right]\forall i, j : i \leq \alpha, j \leq \gamma \qquad (1)$$

- Max-Product: $\mu_T(t_{i,j}) =$

$$\text{Max}\left[\text{Prod}\left(\mu_R(r_{k,i}), \mu_S(s_{j,k})\right)\forall k : \leq \beta\right]\forall i, j : i \leq \alpha, j \leq \gamma \qquad (2)$$

- Sum-Product: $\mu_T(t_{i,j}) =$

$$\sum\left[\text{Prod}\left(\mu_R(r_{k,i}), \mu_S(s_{j,k})\right)\forall k : \leq \beta\right]\forall i, j : i \leq \alpha, j \leq \gamma \qquad (3)$$

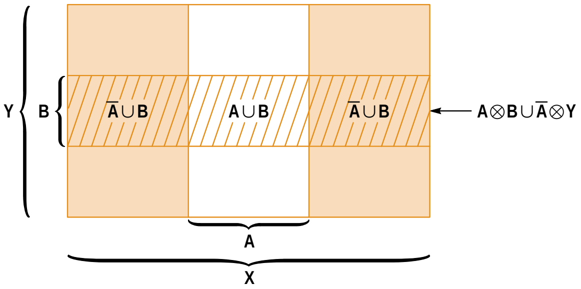


Fig. 9. A graphical depiction of the fuzzy logic implication R = (A ⊗ B) ∪ ($\overline{A}$ ⊗ Y).

where:

α, β, γ are the number of elements (cardinality) in the fuzzy sets X, Y, and Z

i, j, k are subscripts for matrix representations for the fuzzy relations.

The Max-Min composition operator selects the maximum membership value from all the minimal values of every corresponding membership pair. For example, the Max-Min value from the membership pairs [(0.4,0.6), (0.2,0.5)] is 0.4. The Max-Prod composition operator replaces the Min function in the Max-Min operator with the Prod function, which performs algebraic multiplication for every membership pair. The Max-Prod value for our example above is 0.24. Finally, the Sum-Prod operator is derived from the Max-Prod operator by replacing the Max function with the Sum function, which adds together the results of the Prod operations on each membership pair. Applying the Sum-Prod operator to the example above gives the value 0.34. The are many other composition operators available, some of which are designed for specific applications.

Deriving Fuzzy Relations. The most difficult part about developing an application using fuzzy relations is obtaining the relations themselves. Some of the methods used to derive fuzzy relations include:

- Intuitive knowledge, human experience, and opinions of experts
- Calculation methods for membership functions
- Fuzzy compositions
- Converted frequencies and probabilities.

Example. The following example illustrates how to derive a fuzzy relationship. Consider the fuzzy sets and universes described earlier:

Universe Y = {beer, wine, spirits}

Universe Z = {Zirconian, Opalinian, Topazian}

We will assume for this example that the relation R on universe Y **x** Z is based on the opinions of experts who know a lot about the drinking habits of the inhabitants of the provinces contained in universe Z.

$$R = \left.\overbrace{\begin{bmatrix} 0.6 & 0.3 & 0.1 \\ 0.4 & 0.8 & 0.3 \\ 0.2 & 0.1 & 0.7 \end{bmatrix}}^{Z}\right\} Y$$

Although the relation R is derived from intuitive knowledge and experience, we could have used one of the other methods to derive it based on some partial information. Remember that a fuzzy relation captures the pairwise strength of the relation between elements of both universes, which in this case consists of beer, wine, and spirits in rows and Zirconian, Opalinian, and Topazian in columns. For example, there is a strong (0.8) possibility of an Opalinian being a wine lover according to relation R.

Now let's take a beer lover described by the fuzzy set

C = 0.8/beer + 0.2/wine + 0.1/spirits

and perform Max-Min composition on the relation

$H = C \otimes R$

Applying Max-Min yields:

$\mu_H(\text{Zirconian}) =$
$\text{Max}[(\text{Min}(0.8,0.6),\text{Min}(0.2,0.4),\text{Min}(0.1,0.2)] = 0.6$

$\mu_H(\text{Opalinian}) =$
$\text{Max}[(\text{Min}(0.8,0.3),\text{Min}(0.2,0.8),\text{Min}(0.1,0.1)] = 0.3$

$\mu_H(\text{Topazian}) =$
$\text{Max}[(\text{Min}(0.8,0.1),\text{Min}(0.2,0.3),\text{Min}(0.1,0.7)] = 0.2$

Therefore, the resulting fuzzy set H = 0.6/Zirconian + 0.3/Opalinian + 0.2/Topazian might suggest that a beer lover is of predominantly Zirconian heritage with slight linkages to Opalinian influences and very slight Topazian influences based on the experts' opinion represented in relation R.

Fuzzy Family Assignment Heuristic

The goal of our fuzzy family assignment heuristic is to find products with similar components and group them into families. In our family assignment heuristic there are two nested iteration loops: an outer loop for each family being created and an inner loop for selecting the "best-suited" product to assign to the family. The inner loop is terminated when there are no more products to be considered. The outer loop is terminated either when there are no more families or when no more products are being assigned to a particular family. The following is a pseudo-code representation of our algorithm.

```
Family = Primary /* Initialization family variable */
REPEAT            /* Start outer loop              */
  Qualify = PCA /* Products to be assigned to a family.*/
  WHILE Qualify <> Empty /* Start inner loop. Loop  */
                         /* until there are no more products  */
    Find Product from Qualify with the highest selectivity
    measure ($s_i$)
    IF (a qualified Product is selected AND the slots
      required by the selected Product $\leq$ slot availability
      of Family
    THEN
      Assign Product to Family and update slot
      availability of Family
      Remove Product from PCA
    END IF
    Remove Product from Qualify
  END WHILE
  Family = get_a_new_family(Family)
UNTIL (PCA does not change OR no more Families)
```

Product product being considered for inclusion in a family

get_a_new_family returns next available family's name and slot availability

slot availability counter for the number of placement machine slots available to a family (This number is decreased by the number of slots required by each product assigned to the family.)

PCA list of products to be considered for family assignment (This list is updated each time a product is assigned to a family.)

Qualify same as PCA except that this variable is used to determine when to terminate the inner loop and is updated at each iteration.

At the end of this algorithm there still might be products that cannot be assigned to any family because of slot availability, or there might be families with no products assigned.

We used the concepts of fuzzy sets, fuzzy relations, and fuzzy composition to determine which products to select and assign to each family. The variables used in our algorithm include a fuzzy set p_i which represents the printed circuit assembly products, a selectivity measure s_i, which is a value that indicates how each product p_i might fit into a particular family, and finally, the fuzzy relation r, which is used to capture the relation between selectivity s_i and product p_i.

Since the volume, commonality (common parts), and additional slots are the three independent qualifiers that describe a product, they were used to define the product universe P = {commonality, slots, volume}. Thus, a product

$$p_i = m1_i/\text{commonality} + m2_i/\text{volume} + m3_i/\text{slots} \qquad (4)$$

is a fuzzy set on universe P where $m1_i$, $m2_i$, and $m3_i$ are the membership values on the interval <0,1> for product p_i.

We implemented the following computational methods to obtain the membership values for universe P.

- General commonality. General commonality between product p_1 and p_2 is defined as:

$$\text{comm}(p_1,p_2) = \text{ncs}(p_1,p_2)/\text{tns}(p_1) \qquad (5)$$

where ncs is the number of slots common to p_1 and p_2, and tns is the total number of slots required by p_1. It could be deduced that in general:

$$\text{comm}(p_1,p_2) \neq \text{comm}(p_2,p_1) \text{ unless } \text{tns}(p_1) = \text{tns}(p_2).$$

- Commonality during primary family selection:

$$m1_i = \frac{\sum_{j=1}^{j=N \wedge j \neq i} \text{comm}\left(p_j, p_i\right)}{N - 1} \quad \text{for product } p_i. \qquad (6)$$

N is the number of products not yet assigned to a family.

- Commonality during nonprimary family selection:

$$m1_i = \frac{\sum_{j=1}^{j=N \wedge j \neq i} \text{comm}\left(p_i, p_j\right)}{N - 1} \quad \text{for product } p_i. \qquad (7)$$

N is the same as above.

- Volume:

$$m2_i = \frac{\text{demand}\left(p_i\right)}{\underset{j=1}{\overset{j=N}{\text{Max}}}\left(\text{demand}\left(p_j\right)\right)} \quad \text{for product } p_i \qquad (8)$$

where demand(p_i) is the expected volume demand for product p_i and N is the same as above.

- Slots:

$$m3_i = 1 - \frac{\text{slots}\left(p_i\right)}{\text{slots\_available}_t} \quad \text{for product } p_i \qquad (9)$$

where slots(p_i) is the number of additional slots required for product p_i if it is selected, and slots_available_t is the number of slots available for a particular family during iteration t of the assignment algorithm.

Selectivity and Fuzzy Relation

Since the selectivity measure s is defined on the universe S = {selectivity}, the selectivity for product p_i is defined as a fuzzy set on universe S:

$$s_i = m_i/\text{selectivity}. \qquad (10)$$

Fuzzy relation r on universe R = P **x** S is used to capture the relation between product selectivity and the product itself. When we translate the general notion of a fuzzy relation into the reality of our problem, we end up with a 3 × 1 matrix representation of the relation. The column symbolizes the cardinality of universe S and the three rows relate to the product universe P (i.e., commonality, volume, and slots). Since different selection criteria might be desired at different stages of the selection process, we found a need for at least two distinct relations. Thus, based on our experience we selected the following two categories that might require separate fuzzy relations r.

- First product assigned to a primary or nonprimary family
- Nonfirst product assigned to a primary or nonprimary family.

The hardest part about using fuzzy relations is obtaining their membership values. We wanted the membership values derived for the relation r to express the importance assigned to each of the three elements in universe P (i.e., commonality, volume, and slots) during the process of selecting products to add to a particular family. For example the relation:

$$r_i = \begin{bmatrix} 0.7 \\ 0.4 \\ 0.2 \end{bmatrix} \begin{array}{l} \text{(commonality)} \\ \text{(volume)} \\ \text{(slots)} \end{array}$$

says that for product p_i during an iteration of the assignment algorithm, commonality is to be given greater emphasis in family assignment than volume or slots membership values.

One can use one of many fuzzy composition operators to construct the relation, or one can intuitively guess the fuzzy relation r based on some empirical experience or expertise. In our prototypical implementation, we selected the second approach.

Initially, we experimented with the empirically derived graph shown in Fig. 10 to come up with the membership values for the two categories of fuzzy relations mentioned above. Note in Fig. 10 that the fuzzy relationship values for commonality, volume, and slots are dependent on the slots membership value. For example, a slots membership value of 0.5 would provide the relation matrix:

$$r_i = \begin{bmatrix} 0.8 \\ 0.2 \\ 0.0 \end{bmatrix}.$$

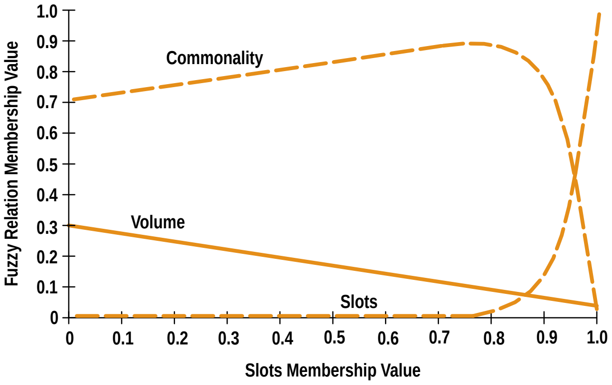


Fig. 10. Fuzzy relation membership values for our experimental functional approach to family assignment.

This approach turned out to be too complex and cumbersome because of our lack of experience with creating membership relations. Consequently, at the end we settled for a constant determination of a relation's individual values to get us started. This approach resulted in the following two fuzzy relations in our prototypical implementation:

- First product assigned to a primary or nonprimary family

$$r_i = \begin{bmatrix} 0.550 \\ 0.150 \\ 0.300 \end{bmatrix} \text{ for product } p_i \ (1 < i < N)$$

- Nonfirst product assigned to a primary or nonprimary family

$$r_i = \begin{bmatrix} 0.550 \\ 0.200 \\ 0.250 \end{bmatrix} \text{ for product } p_i \ (1 < i < N)$$

where N is the number of products not yet assigned.

It is important to notice that in general no restrictions are imposed on fuzzy membership values, but since we used the Sum-Product fuzzy composition operator then the summation of the elements in each relationship matrix must be ≤ 1.

Example

The fuzzy composition for our family assignment problem is $s_i = r_i \otimes p_i$. Although we investigated a number of fuzzy composition operators, we had the most success with the Sum-Product composition operator.

The following example illustrates the actions performed by the assignment algorithm to select a product to assign to a particular family.

- Assume there are three products p_1, p_2, and p_3 and that we are selecting the first product to be assigned to a primary or nonprimary family.
- ncs (number of common slots) for pairs of p_i, p_j for $i \leq 3$, $j \leq 3$ is:

$$\begin{bmatrix} 20 & 10 & 12 \\ 10 & 30 & 21 \\ 12 & 21 & 40 \end{bmatrix}$$

- slots_available = 45
- demand $(p_1) = 85$, additional slots$(p_1) = 20$

From equation 4:

$$p_1 = m1_1/\text{commonality} + m2_1/\text{volume} + m3_1/\text{slot}$$

where:

by equation 5: $\text{comm}(p_1, p_2) = 10/20 = 0.5$ and
$\text{comm}(p_1, p_3) = 12/20 = 0.6$.

From equations 7, 8, and 9:

$$m1_1 = \frac{0.6 + 0.5}{2} = 1.1/2 = 0.55$$
$$m2_1 = 85/\text{Max}(85, 100, 60) = 0.85$$
$$m3_1 = 1 - 20/45 = 0.56.$$

Finally,

$$p_1 = 0.55/\text{commonality} + 0.85/\text{volume} + 0.56/\text{slot}$$

- demand$(p_2) = 100$, additional slots$(p_2) = 30$, and

$$p_2 = 0.51/\text{commonality} + 1.00/\text{volume} + 0.33/\text{slot}$$

- demand$(p_3) = 60$, additional slots$(p_3) = 40$, and

$$p_3 = 0.41/\text{commonality} + 0.60/\text{volume} + 0.11/\text{slot}.$$

- Since $r_{1,2,3} = \begin{bmatrix} 0.550 \\ 0.150 \\ 0.300 \end{bmatrix}$ using equation 3, the Sum-Product operator, the fuzzy composition $s_i = r_i \otimes p_i$ for this example is:

$$s_1 = r_1 \otimes p_1 = 0.55 \times 0.55 + 0.150 \times 0.85 + 0.300 \times 0.56 = 0.598$$

$$s_2 = r_2 \otimes p_2 = 0.529$$

$$s_3 = r_3 \otimes p_3 = 0.348.$$

- Finally, since $\text{Max}(s_1, s_2, s_3) = s_1$, product p_1 has the highest selectivity measure and is therefore assigned to the family being formed during this iteration.

Fuzzy Machine Balancing

After the fuzzy family assignment algorithm assigns the products to their corresponding families, the fuzzy machine balancer tries to assign each family's components to the placement machines. The primary objective is to have each side of the assembled printed circuit assembly use the two series CP-IIIs as equally as possible.

Constraints

Aside from the inherent constraints introduced by families, manufacturing reality brings a few special cases of already predetermined machine assignments and constraints.

Physical Process Constraints. Since the objective is to have as much setup slot room as possible, certain physical process related limitations arise. For example, constraints on the very last slot available on the bank do not allow a two-slot-wide feeder to be mounted on the last slot. If we have one slot still available on each machine and we have to place a two-slot-wide feeder, we need to move a one-slot-wide feeder from one machine to make room for the two-slot-wide feeder. Finally, a component whose package is higher than 3.5 mm must be placed by the second machine since the component height might interfere with the placing nozzle on a densely populated printed circuit assembly.

Primary Family Products. If the printed circuit assembly members in the primary family could be balanced without consideration for the remaining printed circuit assemblies that use a portion of the primary family, balancing could probably be achieved at the expense of the remaining products' imbalance. Thus, it is crucial that balancing for primary family products take into consideration the remaining products.

Nonprimary Family Products. In the case of nonprimary family products, the problem is just the opposite of the problem encountered for primary family products. For nonprimary family products balancing has to incorporate the component assignments already committed by the primary family balancing procedure.

Placement Time Estimation. The true placement time for a printed circuit assembly is a function of the placement sequence, which includes the placement table movement, the placing head rotation speed, and the feeder bank movement. The only information we have available is the placing head rotation speed and even that is an approximation. The maximum allowable speed for the placing head rotation is determined from a component's polarity, presentation, package type, size, and pickup nozzle size. Furthermore, the placing head has 12 two-nozzle stations that are all influenced by the head's speed selection. We approximated the placement speed by obtaining the speed of the head rotation.

Products with Inherent Imbalance. In certain cases only the duplication of a component's availability among the CP-III placement machines would lead to good balance. For example, it is possible that a printed circuit assembly's side requires a placement-intensive component that greatly exceeds the total placement of the remaining components. Only an availability of that component in both of the CP-III setups would provide a shot at a reasonable balance. In our initial implementation we didn't use this approach.

Algorithm Outline

Just as in our family assignment approach we used the concepts of fuzzy sets, relationships, and fuzzy composition to balance the series CP-III loads for each printed circuit assembly side being assembled. The following is a high-level procedural outline of our balancing algorithm.

```
Define component fuzzy set $c_i$ for $1 < i < N$
Sort all $c_i$ in decreasing order
Initialize fuzzy relation r
FOR $1 < i < N$
  IF $c_i$ has no predetermined matching assignment $m_a$
  THEN
    $m_i = r \otimes c_i$
    $m_i$ Defuzzification $\Rightarrow m_a$ for $c_i$
  END IF
  Assign component $c_i$ to machine $m_a$
  Update the relation r; $r = \text{rel\_update}(c_i, m_a)$
END FOR
Ensure that all machine constraints are satisfied.
```

c_i is the ith component represented by the fuzzy set c
N is the number of components to be assigned
m_i is the machine fuzzy set obtained for component c_i
m_a is an actual machine a to which the component c_i has been assigned
r describes the relation between c_i and m_i
$\otimes$ is the fuzzy operator.

Nonfuzzy sorting of fuzzy sets is based on $\sum_{i=1}^{N} W_{i,j}$ where $W_{i,j}$ is a value described for all fuzzy components c_i (described below), i is the ith component, and j is the jth product.

Fuzzy Component c

A fuzzy set c_i representing a physical component C_i is defined on universe $P = \{p_1, p_2, \ldots p_q\}$ where $p_1, p_2, \ldots p_q$ represent products. Thus, fuzzy set

$$c_i = \left(w_{i,1}/p_1,\ w_{i,2}/p_2,\ \ldots\ w_{i,j}/p_q\right) \tag{11}$$

where:

$$w_{i,j} = W_{i,j}/\text{norm}(C_i) \tag{12}$$

$$W_{i,j} = \text{w\_place}(C_i) \times \text{qty\_per}(C_i,p_j) \times \text{no\_images}(p_j) \times \log_{10}(\text{demand}(p_j)) \tag{13}$$

w_place is a placement time weight factor for physical component C_i
qty_per is the number of times a component C_i is placed on product p_i
no_images is the number of times product p_i appears on a single manufacturing fixture (panel)
demand is the expected volume demand for product p_i

$$\text{norm}(C_i) = \operatorname*{Max}_{j=1}^{Q}\left(W_{i,j}\right) \tag{14}$$

Q is the cardinality of universe P.

Machine Fuzzy Set m

The machine fuzzy set m is defined on the universe M = {CP3.1,CP3.2}. Consequently, the fuzzy set m_i is defined as a fuzzy set on universe M as:

$$m_1 = w_{i,1}/\text{CP3.1} + w_{i,2}/\text{CP3.2} \tag{15}$$

and it is obtained by $r_t \otimes c_i$, where $\otimes$ is the fuzzy composition operator of choice.

Fuzzy Relation r

Fuzzy relation r on universe $R = P \mathbf{x} M$ is used to capture the relation between the physical component C represented by fuzzy set c and machine fuzzy set m. We developed the following general equation to obtain membership values for the relation r.

$$r_{k,n} = 1 - \text{assigned\_current}_{k,n}/\text{assigned\_expected}_{k,n} \tag{16}$$

where:

$0 < k < Q$ since Q is the cardinality of universe P
$1 < n < 2$ since universe M has two elements CP3.1 and CP3.2
$\text{assigned\_current}_{k,n}$ is the current assignment for the kth product and the nth physical machine
$\text{assigned\_expected}_{k,n}$ is the expected assignment for the kth product and the nth physical machine.

If $\text{assigned\_current}_{k,n} > \text{assigned\_expected}_{k,n}$ then $r_{k,n} = 0$ ($r_{k,n}$ should be in the <0,1> interval).

We considered two possible ways to obtain values for $\text{assigned\_current}_{k,n}$ and $\text{assigned\_expected}_{k,n}$. In the first approach, we only considered the component placement time without any additional consideration for slot space limitations. In the second approach, we tried to incorporate some of the known slot constraints. The following equations show the two approaches for obtaining the values for $\text{assigned\_current}_{k,n}$ and $\text{assigned\_expected}_{k,n}$:

- Placement time only.

$$\text{assigned\_expected}_{k,n} = \left[\sum_{j=1}^{j=N \wedge C_j \notin A} \left(W_{j,k}\right) + ac_{k,n}\right] \times pp_{k,n} \qquad (17)$$

$$\text{assigned\_current}_{k,n} = \sum_{j=1}^{z=N \wedge C_j \in \text{Mach}_n} \left(W_{j,k}\right) + ac_{k,n} \qquad (18)$$

where:

$ac_{k,n}$ is the placement time sum for components committed to the nth machine for the kth product

$pp_{k,n}$ is the percentile portion of the kth product preferred to be consumed at the nth physical machine

Mach_n is a crisp set of all physical components C assigned to the nth physical machine

$$A = \bigcup_{n=1}^{n=2} \text{Mach}_n \qquad (19)$$

$W_{j,k}$ see the definition of the fuzzy component c given above

N is the number of components to be assigned.

- Placement time with slot constraints considered.

$$\text{assigned\_expected}_{k,n} = \left[\left[\sum_{j=1}^{j=N \wedge C_j \notin A} \left(W_{j,k} \times s_j\right)\right] \times \text{avail}_n + ac_{k,n}\right] \times pp_{k,n}$$

$$\text{assigned\_current}_{k,n} = \left(\sum_{j=1}^{j=N \wedge C_j \in \text{Mach}_n} \left(W_{j,k} \times s_j\right)\right) \times \text{taken}_n + ac_{k,n}$$

where:

$ac_{k,n}$ is the placement time sum for components committed to the nth machine for the kth product

$pp_{k,n}$ is the percentile portion of the kth product preferred to be consumed at the nth physical machine

Mach_n is a crisp set of all physical components C assigned to the nth physical machine

$W_{j,k}$ see the definition of the fuzzy component c given above

s_j is the number of slots component C_j consumes

avail_n is the number of slots available for the nth machine

taken_n is the number of slots already taken at the nth machine

$A = \bigcup_{n=1}^{n=2} \text{Mach}_n$ is a crisp set containing components already assigned to some machine.

Note that the $ac_{k,n}$ identifier used in both approaches includes the predetermined components already assigned to a machine. Thus, the fuzzy machine balancer does not actively consider predetermined components for balancing, it simply incorporates their passive balancing impact into the balancing process.

The second approach is very complex and elaborate and tries to control a lot of independent measures simultaneously. Thus we selected the first approach for our prototype because we achieved much better overall balance with this approach even though we have to ensure that slot constraints are satisfied in an independent postbalancing step.

The fuzzy relation r has to be updated every time a component C is assigned to the nth physical machine. This procedure ensures that the current component assignment is going to be reflected by the fuzzy relation r. This update is done fairly quickly by recalculating the $\text{assigned\_current}_{k,n}$ value for the corresponding machine n and product p_k ($0 < k < Q$). It is obvious that the update of $\text{assigned\_current}_{k,n}$ changes the appropriate $r_{k,n}$ and hence the fuzzy relation r.

Fuzzy Composition

Although the Max-Min and Sum-Prod composition operators were investigated, the Max-Prod fuzzy composition operator performed best for our balancing algorithm, and we used the Max defuzzification approach to select a component to assign to a particular machine.

The following example illustrates our machine balancing algorithm. In this example we are trying to assign component c_{10}, and we have three products p_1, p_2, and p_3 to assemble. Components c_1 to c_9 have already been assigned to one of two placement machines CP3.1 and CP3.2

To simplify our calculations assume that $ac_{k,n}$ is 0 for all k and all n meaning there are no predetermined components on any of the products in question.

From equation 11:

$$c_{10} = w_{10,1}/p_1 + w_{10,2}/p_2 + w_{10,3}/p_3$$

and from equation 12

$$w_{10,j} = W_{10,j}/\text{norm}\left(C_{10}\right)$$

If we assume that $W_{10,(1,2,3)} = (112.72, 150.0, 0.0)$ then using equations 12 and 14:

$$w_{10,1} = 112.72/150.0 = 0.75$$
$$w_{10,2} = 150.0/150.0 = 1.0$$
$$w_{10,3} = 0.0.$$

Thus,

$$c_{10} = 0.75/p_1 + 1.0/p_2 + 0.0/p_3.$$

Assume that after computing equations 17 and 18 we get the following values for each placement machine:

$$\text{assign\_expected} = \begin{array}{c} \begin{array}{cc} \text{CP3.1} & \text{CP3.2} \end{array} \\ \begin{bmatrix} 1713.0 & 1713.0 \\ 2000.0 & 2000.0 \\ 459.0 & 459.0 \end{bmatrix} \end{array} \begin{array}{l} \\ p_1 \\ p_2 \\ p_3 \end{array}$$

and

$$\text{assign\_current} = \begin{bmatrix} 1273.0 & 498.0 \\ 1782.0 & 1560.0 \\ 150.0 & 450.0 \end{bmatrix}.$$

The assign_expected matrix has the same values in both columns because we want to balance the load equally between the two placement machines for products p_1, p_2, and p_3. Assign_current shows the component balance between the two machines for components c_1 through c_9 at the current iteration of the balancing algorithm.

Using equation 16

$$r = \begin{bmatrix} 0.26 & 0.72 \\ 0.11 & 0.21 \\ 0.67 & 0.02 \end{bmatrix}.$$

Referring to steps 6, 7, 8, and 9 in our machine balancing algorithm, the following items are computed.

- Step 6. From equation 14, $m_i = r_t \otimes c_i$ using the Max_Prod fuzzy composition operator in equation 2:

$$r_{10} \otimes c_{10} = \text{Max}\left\{ \begin{bmatrix} 0.75 & 1.0 & 0.0 \end{bmatrix} \begin{bmatrix} 0.26 & 0.72 \\ 0.11 & 0.21 \\ 0.67 & 0.02 \end{bmatrix} \right\}$$

$$= \text{Max}\left[(0.19 \;\; 0.11 \;\; 0.0)(0.54 \;\; 0.21 \;\; 0.0) \right]$$

Thus,

$$m_{10} = 0.19/\text{CP3.1} + 0.54/\text{CP3.2}$$

- Steps 7 and 8. Defuzzification ⇒ m_a for c_i is obtained by applying the Max defuzzification method to m_{10}. Thus, $\text{Max}(m_{10}) \Rightarrow m_2 = \text{CP3.2}$ meaning that component c_{10} is assigned to machine CP3.2 since it has the maximal membership value (0.54).

- Step 9. Update the relation r.

$r = \text{rel\_update}(c_i, m_a)$ and updating

$$\text{assign\_current} = \begin{bmatrix} 1273.0 & 601.72 \\ 1782.0 & 1710.0 \\ 150.0 & 450.0 \end{bmatrix} \text{ at } i = 10$$

makes

$$r = \begin{bmatrix} 0.26 & 0.64 \\ 0.11 & 0.15 \\ 0.67 & 0.02 \end{bmatrix}$$

for the next iteration.

Results

For this experiment we used two manufacturing production lines at our site. The first one is denoted as line 1 and the second one as line 2. The total line volume is equivalent between the two lines. The statistics on the two lines include:

- Line 1: 27 products, 13 double-sided, 413 unique components, and on average a component appears on 3.05 products.
- Line 2: 34 products, 11 double-sided, 540 unique components, and on average a component appears on 4.69 products.

Fig. 11 shows the setup families created for the printed circuit assembly products assigned to lines 1 and 2.

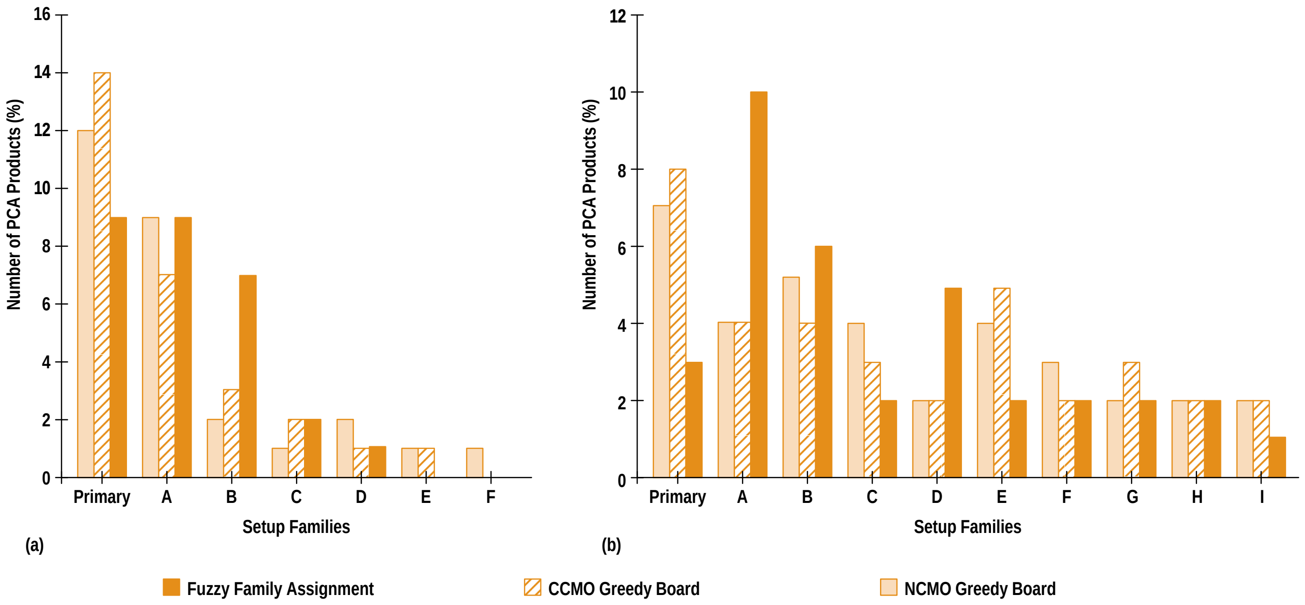


Fig. 11. The setup families created and the number of printed circuit assembly products contained in each family based on the type of family assignment algorithm used. (a) Line 1. (b) Line 2.

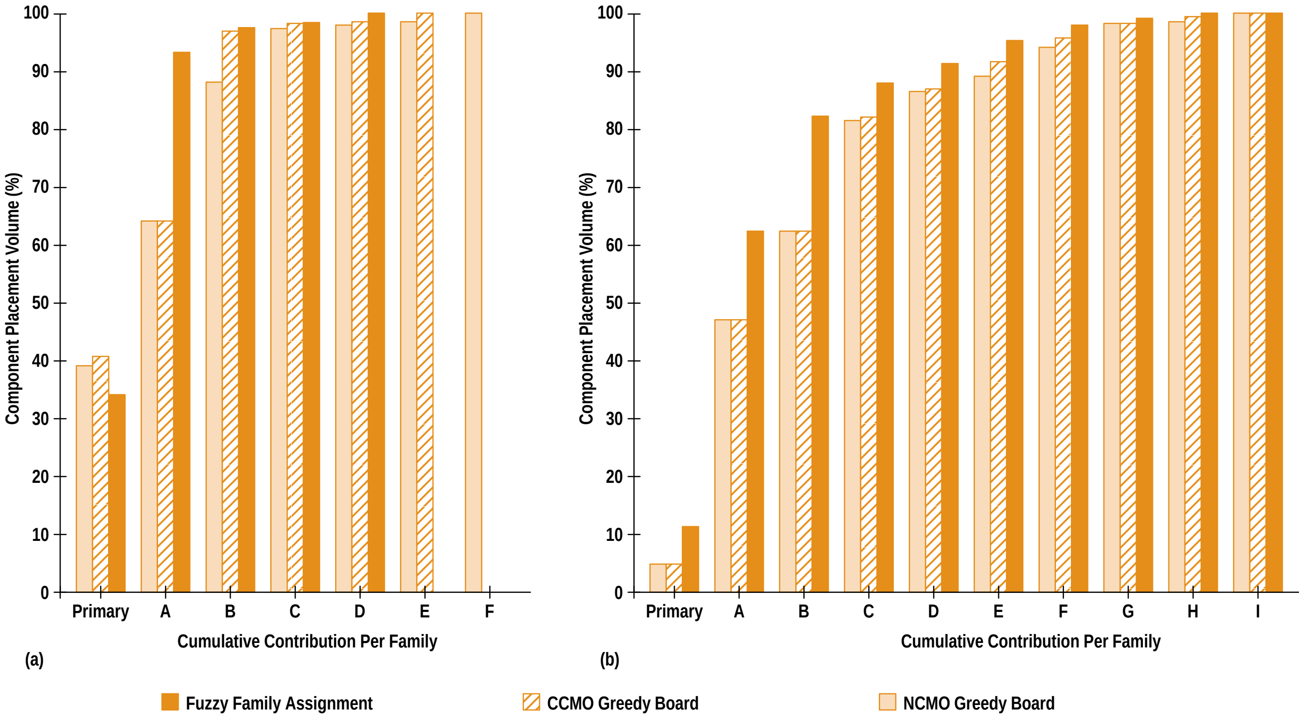


Fig. 12. A cumulative representation of the family assignments for line 1 and line 2 versus component placement volume. (a) Line 1. (b) Line 2.

Family Assignment

Fig. 12 shows the percentage of component placement volume versus the cumulative contribution for each of the family assignment techniques described in this paper.

Line 1. The results for this line were indeed phenomenal. The primary and A families together constitute 95% of component placement volume for the line. This results in no need for setup changeover for 95% of the volume for one

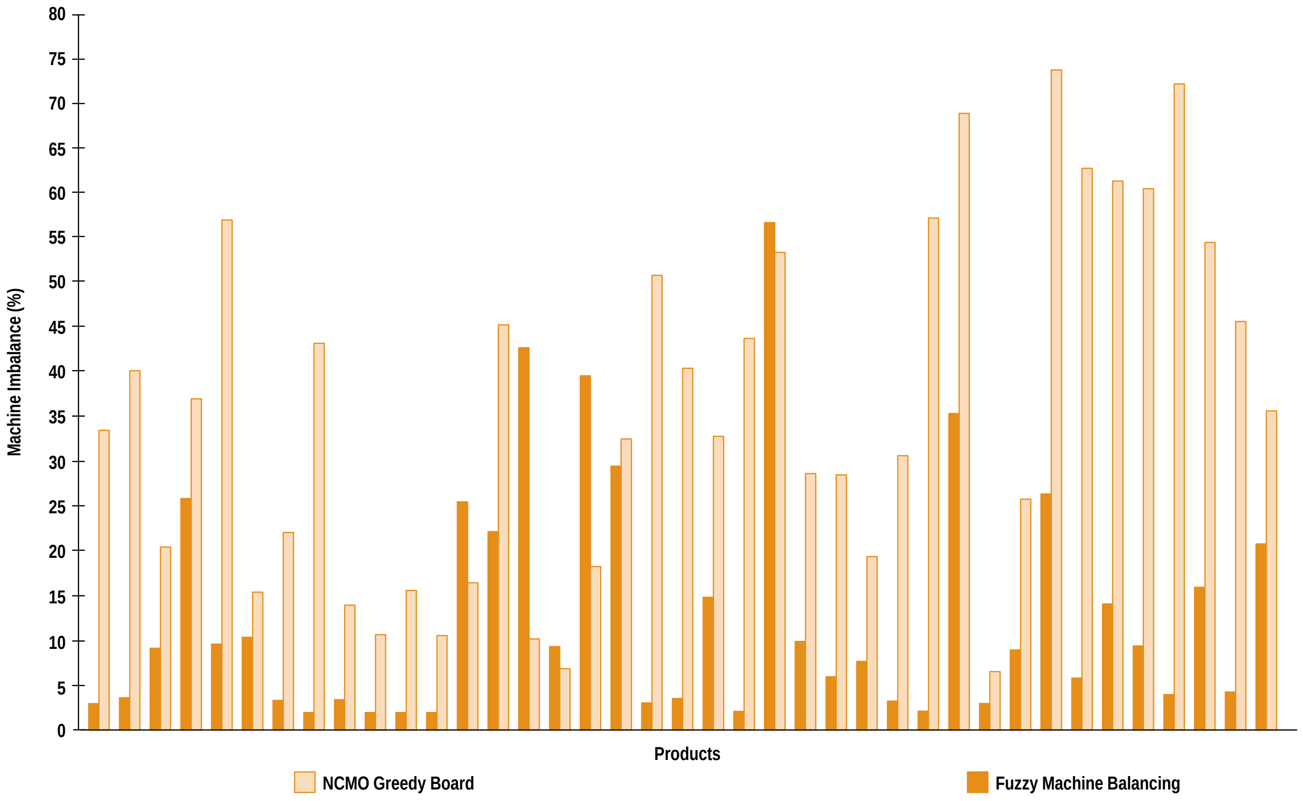


Fig. 13. Machine imbalance for line 1.

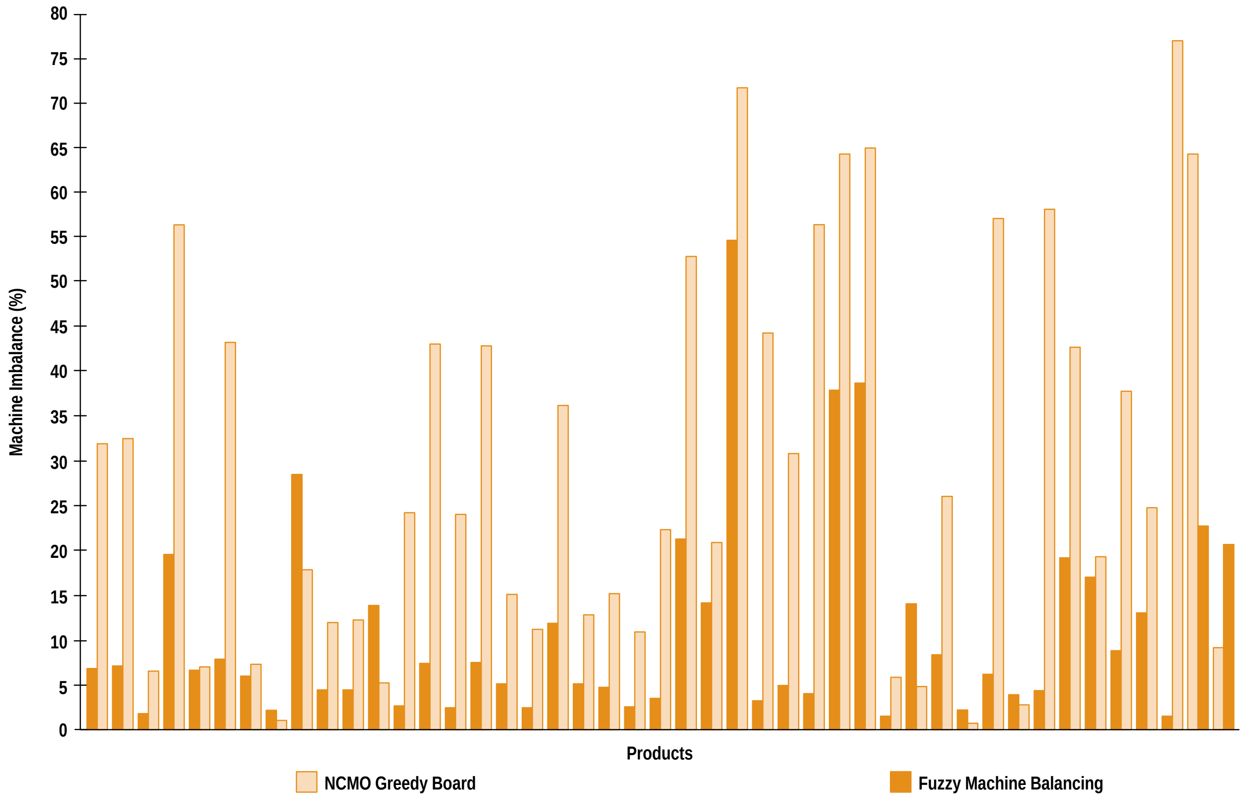


Fig. 14. Machine imbalance for line 2.

month. The 28% reduction of the number of families is a side effect of the fuzzy family assignment optimization.

Line 2. The major achievement of the fuzzy family assignment technique for line 2 was not just the moderate volume improvements over the greedy board and CCMO greedy board, but its ability to produce the same solution we obtained when we manually forced certain products into a primary family using the greedy board method. When we first investigated greedy board capabilities, we allowed hand-picked products to be forced into a family, regardless of their greedy ratio. The forced products were carefully identified based on our intuition and expertise.

Machine Balancing

Figs. 13 and 14 show the percentage imbalance for individual printed circuit assemblies manufactured on lines 1 and 2 respectively. The line 1 average imbalance was 12.75% for the fuzzy machine balancing approach and 35.9% for the balance obtained by the greedy board approach. The line 2 results are 10.73% for the fuzzy machine balancing approach and 29.58% for the greedy board approach. The families are the same ones provided by the fuzzy family assignment method.

Acknowledgment

Although many of my colleagues at HP CCMO have frequently discussed the problems described in this paper with me, I am specially grateful to my colleague Jim Nussbaumer for a lot of informal but exceptionally fruitful discussions, which have brought simplicity and elegance into our fuzzy family assignment and machine balancing approach.

References

1. T. Davis and E. Selep, “Group Technology for High-Mix Printed Circuit Assembly,” *IEEE International Electronic Manufacturing Technology Symposium,* October, 1990.

2. M Jamshidi, N. Vadiee, and T. Ross, *Fuzzy Logic and Control: Software and Hardware Applications,* Prentice Hall, 1993.