The Network Architecture of the Connection Machine CM-5

Charles E. Leiserson,[*] Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul,[†] Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak

Thinking Machines Corporation
Cambridge, Massachusetts 02142

November 9, 1992

Abstract

The Connection Machine Model CM-5 Supercomputer is a massively parallel computer system designed to offer performance in the range of 1 teraflops (10^{12} floating-point operations per second). The CM-5 obtains its high performance while offering ease of programming, flexibility, and reliability. The machine contains three communication networks: a data network, a control network, and a diagnostic network. This paper describes the organization of these three networks and how they contribute to the design goals of the CM-5.

1 Introduction

In the design of a parallel computer, the engineering principle of *economy of mechanism* suggests that the machine should employ only a single communication network to convey information among the processors in the system. Indeed, many parallel computers contain only a single network: typically, a hypercube or a mesh. The Connection Machine Model CM-5 Supercomputer has three networks, however, and none is a hypercube or a mesh. This paper describes the architecture of each of these three networks and the rationale behind them.

Figure 1 shows a diagram of the the CM-5 organization. The machine contains between 32 and 16,384 *processing nodes*, each of which contains a 32-megahertz SPARC processor, 32 megabytes of memory, and a 128-megaflops vector-processing unit capable of processing 64-bit floating-point and integer numbers. System administration tasks and serial user tasks are executed by a collection of *control processors*, which are Sun Microsystems workstation computers. There are from 1 to several tens of control processors in a CM-5, each configured with memory and disk according to the customer's preference. Input and output is provided via high-bandwidth *I/O interfaces* to graphics devices, mass secondary storage, and high-performance networks. Additional low-speed I/O is provided by Ethernet connections to the control processors. The largest machine, configured with up to 16,384 processing nodes, occupies a space of approximately 30 meters by 30 meters, and is capable of over a teraflops (10^{12} floating-point operations per second).

The processing nodes, control processors, and I/O interfaces are interconnected by three networks: a data network, a control network, and a diagnostic network. The data network provides high-performance point-to-point data communications between system components. The control network provides cooperative operations, including broadcast, synchronization, and *scans* (parallel prefix and suffix). It also provides system management operations, such as error reporting. The diagnostic network allows "back-door" access to all system hardware to test system integrity and to detect and isolate errors.

The system operates as one or more user *partitions*. Each partition consists of a control processor, a collection of processing nodes, and dedicated portions of the data and control networks. Access to system functions is classified as either *privileged* or *nonprivileged*. All nonprivileged system functions, including access to the data and control networks, can be executed directly by user code without system calls. Consequently, network communication within a user task occurs without operating system overhead. Access to the diagnostics network, to shared system resources (such as I/O), and to other partitions is privileged and must be accomplished via system calls. Protection and

[*]Charles E. Leiserson is a Corporate Fellow of Thinking Machines Corporation and Professor of Computer Science and Engineering in the MIT Laboratory for Computer Science.

[†]Bradley C. Kuszmaul is a consultant to Thinking Machines Corporation and a graduate student in the MIT Laboratory for Computer Science.

An early version of this paper appeared in the 1992 ACM Symposium on Parallel Algorithms and Architectures.

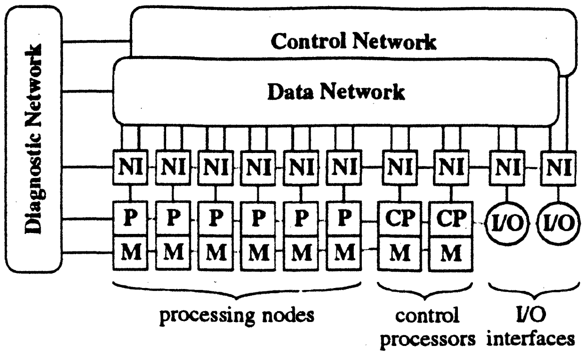

Figure 1: The organization of the Connection Machine CM-5. The machine has three networks: a data network, a control network, and a diagnostic network. The data and control networks are connected to processing nodes, control processors, and I/O channels via a network interface.

addressing mechanisms ensure that no user can interfere with the function or performance of another user in another partition. If the system administrator so desires, a single partition can be timeshared among a group of users, where each user gets a fair portion of the available time and cannot otherwise be interfered with by any other user.

Further details about the CM-5 system can be found in the CM-5 Technical Summary [24].

When we first set about designing the CM-5, we established engineering goals that went beyond mere performance specifications. We thought hard about issues of *scalability*: making a machine whose size would be limited only by the dollars a customer could spend, not by any architectural or engineering constraint. We thought hard about system issues, including timesharing, I/O, and user protection. We thought hard about reliability, since we were designing a machine which, in its largest configuration, would have well over 10 times the electronics of our previous supercomputer, the Connection Machine Model CM-2 Supercomputer.

The following goals drove our network designs:

- The networks must deliver high performance to the users. We wanted the users to be easily able to program the networks to get good performance. We did not want to force the users to worry constantly about pathological worst cases, and we wanted the best cases to run well without the user needing to do anything special.
- The networks must scale up to a very large size. We wanted the logical design of the networks to scale up to a million processing nodes. We wanted to build SUPERcomputers.
- The networks should efficiently support the data-parallel programming model (see Section 5), but should be flexible enough to allow us to support other parallel programming models as well. The data-parallel programming model was used extensively on the CM-2, and we wanted to be able to transport our existing high-level programming environments (Fortran90, *Lisp, and C*) to the CM-5. We also wanted to be able to run codes written for other machines competitively.
- The networks must be highly reliable and highly available. The system must notice whenever part of a network fails, be able to isolate the failure quickly, and be able to quickly reconfigure the networks around the failure. It was desired that even if part of a network has failed, the rest of the network should be able to function correctly with only a small degradation in performance.
- The networks must work in a spaceshared environment. We wanted a user's network traffic to be insulated from other users and I/O in other partitions.
- The networks must work in a timeshared environment. A timeshared user must get a fair share of network bandwidth. Users must be able to be context-swapped quickly. Privileged system software must be able to seize control of a user's task.
- The networks must be operational as soon as possible. Time to market was of the essence. Chips and systems needed to work the first time. We wanted the networks to be simple enough to engineer quickly, robust enough to respond to last-minute design changes, and easily verifiable. Consequently, we opted for conservative technology, for example, copper wires rather than optical fibers. We chose to use CMOS in order to minimize the risk associated with new technology. We chose standard-cell technology in order to be able to make extensive use of the wide variety of available design tools (such as timing verifiers and automatic test generators). To achieve high performance with this conservative technology, we incorporated custom macro cells for circuits on the critical

path. Our attitude was that there was more performance to be gained by architectural improvements than by eking out extra nanoseconds in technology. Conservative technologies, with their well-developed computer-aided design tools, would allow us to make many more architectural improvements during the design.

- The chips used to build the networks must be organized in a way to allow technological or architectural improvements to be easily incorporated in subsequent revisions of the CM-5 system. On the CM-2, both processors and communication were implemented on the same chip, which made it difficult to incorporate advanced technology in one area without impacting the other. We wanted to be able to incorporate any advances without having to reengineer a major piece of the system.
- The networks should embody both economy of mechanism and single-minded functionality. We wanted the networks to be lean and mean: Whenever anyone suggested anything complicated, we viewed it with suspicion. For example, the job of the data network is to deliver messages, nothing else. But it delivers both user messages and messages to I/O devices using the same mechanisms. The data network does not combine messages, duplicate messages, or acknowledge delivery of messages. It just focuses on moving data as fast as possible.

We ended up designing two networks visible to the user and a network interface that provides an abstract view of them. We also designed a diagnostic network to provide "back-door" access to the system. This paper describes the three networks, the network interface, and how we engineered them to meet our goals. The reader should be aware that the performance specifications quoted in this paper apply only to the initial release of the CM-5 system. Because of our ability to reengineer pieces of the system easily, these numbers represent only a snapshot of an evolving implementation of the architecture.

The remainder of this paper is organized as follows. To begin, Section 2 describes the network interface which provides the user's view of the data and control networks. Section 3 then describes the data network. A justification for having both a data and control network is provided in Section 4. The control network is then described in Section 5, and Section 6 describes the diagnostic network. The paper closes with Section 7, which gives a short history of our development project.

2 The CM-5 Network Interface

Early on in the design of the CM-5, we decided to specify an interface between the processing nodes and the networks that isolates each from the details of the other. This interface provides three features. First, the interface gives the processors a simple and uniform view of the networks (and the networks get a simple and uniform view of the processors). Second, the interface provides support for time-sharing, space-sharing, and mapping out of failed components. Third, the interface provides a contract for the implementors which decouples the design decisions made for the networks from those of the processors.

The processor's view of the interface is as a collection of memory-mapped registers. By writing to or reading from fixed physical memory addresses, data is transferred to or from the networks, and the interface interprets the particular address as a command.

A memory mapped interface allows us to use many of the memory-oriented mechanisms found in off-the-shelf processors to deal with network interface issues. To access the network, a user or compiler reads from or writes to locations in memory. We regarded the prospect of executing a system supervisor call for every communication as unacceptable, in part because we wished to support the fine-grain communication needs of data-parallel computation. A memory-mapped interface allows the operating system to deny users access to certain network operations by placing the corresponding memory-mapped registers on protected pages of the processor's address space. The processor's memory management unit enforces protection without any additional hardware.

The interface is broadly organized as a collection of memory-mapped FIFO's. Each FIFO is either an *outgoing* FIFO to provide data to a network, or an *incoming* FIFO to retrieve data from a network. Status information can be accessed through memory-mapped registers. For example, to send a message over a network, a processor pushes the data into an outgoing FIFO by writing to a suitable memory address. When a message arrives at a processor, the event is signaled by interrupting the processor, or alternatively, the processor can poll a memory-mapped status bit. The data in the message can then be retrieved by reading from the appropriate incoming FIFO. This paradigm is identical for both the data and control networks.

The network interface provides the mechanisms needed to allow context switching of user tasks. Each user partition in the CM-5 system can run either batch jobs or a timesharing system. When a user is swapped out during timesharing, the processors must save the computation state. Some of this state information is retrieved from

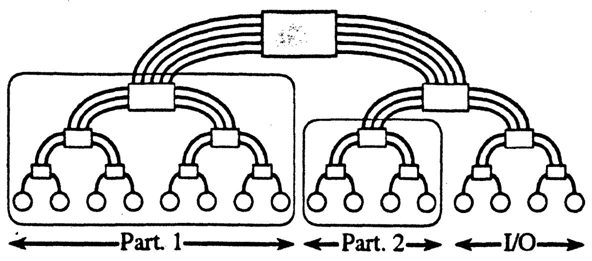

Figure 2: A binary fat-tree. Processors are located at the leaves, and the internal nodes are switches. Unlike an ordinary binary tree, the channel capacities of a fat-tree increase as we ascend from leaves to root. The hierarchical nature of a fat-tree can be exploited to give each user partition a dedicated subnetwork which cannot be interfered with by any other partition's message traffic. The CM-5 data network uses a 4-ary tree instead of a binary tree.

the network interface, and the rest is garnered from the networks. The context-switching mechanism also supports automatic checkpointing of user tasks.

The interface provides processor-address mapping so that the user sees a 0-based contiguous address space for the processor numbers within a partition. Each processor can be named by its *physical* address or by its *relative* address within the partition. A physical address is the actual network address as interpreted by the hardware inside the networks. A relative address gives the index of a processor relative to the start of a user partition, where failed processors are mapped out. All processor addresses in user code are relative addresses. To specify physical addresses requires supervisor privileges. Relative addresses are bounds checked, so that user code cannot specify addresses outside its partition.

The user's view of the networks is independent of a network's topology. Users cannot directly program the wires of the networks, as they could on our previous machine, the CM-2. The reason is simple: the wires might not be there! Because the CM-5 is designed to be resilient in the presence of faults, we cannot allow the user to rely on a specific network topology. One might think topology independence would hurt network performance, but we found this presumption to be less true than we initially imagined. Because we did not provide the user with access to the wires of the network, we were able to apply more resources to generic network capabilities. A further advantage of topology independence is that the network technology becomes decoupled from processor technology. Any future network enhancements are independent of user code and processor organization.

An important ramification of the decoupling of the processors from the networks is that the networks must assume full responsibility for performing their functions. The data network, for example, does not rely on the processors to guarantee end-to-end delivery. The processors assume that delivery is reliable; nondelivery implies a broken system, since there is no protocol for retransmission. By guaranteeing delivery, additional error-detection circuitry must be incorporated into the network design, which slightly reduces its performance, but since the processor does not need to deal with possible network failures, the overall performance as seen by a user is much better.

The CM-5 network interface is implemented in large measure by a single 1-micron standard-cell CMOS chip, with custom macro cells to provide high-performance circuits where needed. The interface chip is clocked by both the 32-megahertz processor clock and the 40-megahertz networks clock. Asynchronous arbiters synchronize the processor side of the interface with the network side.

Choosing to build a separate network interface allowed the processor designers to do their jobs and the network designers to do theirs with a minimum of interference. As a measure of its success in decoupling the networks from the processor organization, the same interface chip is used to interface the network to I/O channels, of which there are many types, including CMIO, VME, FDDI, and HIPPI.

3 The CM-5 Data Network

The basic architecture of the CM-5 data network is a *fat-tree* [8, 16]. Figure 2 shows a binary fat-tree. Unlike a computer scientist's traditional notion of a tree, a fat-tree is more like a real tree in that it gets thicker further from the leaves. Processing nodes, control processors, and I/O channels are located at the leaves of the fat-tree. (For convenience, we shall refer to all of these network addresses simply as processors.)

A user partition corresponds to a subtree in the network. Messages local to a given partition are routed within the partition's subtree, thereby requiring no bandwidth higher in the tree. Access to shared system resources, such as I/O, is accomplished through the part of the fat-tree not devoted to any partition. Thus, message traffic within

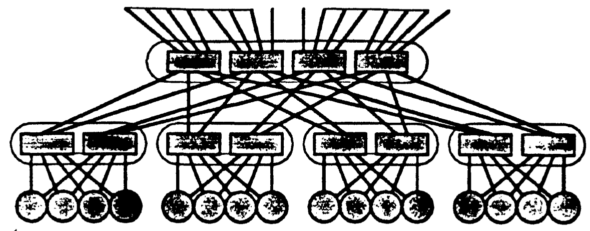

Figure 3: The interconnection pattern of the CM-5 data network. The network is a 4-ary fat-tree in which each internal node is made up of several router chips. Each router chip is connected to 4 child chips and either 2 or 4 parent chips.

a partition, between a partition and an I/O device, or between I/O devices does not affect traffic within any other partitions. Moreover, since I/O channels can be addressed just like processing nodes, the data network becomes a true "system bus" in which all system components have a unique physical address in a single, uniform name-space.

Of critical importance to the performance of a fat-tree routing network is the communication bandwidth between nodes of the fat-tree. Most networks that have been proposed for parallel processing, such as meshes and hypercubes, are inflexible when it comes to adapting their topologies to the arbitrary bandwidths provided by packaging technology. The bandwidths between nodes in a fat-tree, however, are not constrained to follow a prescribed mathematical formula. A fat-tree can be adapted to effectively utilize whatever bandwidths make engineering sense in terms of cost and performance. No matter how the bandwidths of the fat-tree are chosen, provably effective routing algorithms exist [8, 15] to route messages near-optimally. The underlying architecture and mechanism for addressing is not affected by communication bandwidths: to route a message from one processor to another, the message is sent up the tree to the least common ancestor of the two processors, and then down to the destination.

Because of various implementation trade-offs—including the number of pins per chip, the number of wires per cable, and the maximum cable length—we designed the CM-5 data network using a 4-ary fat-tree, rather than a binary fat-tree. Figure 3 shows the interconnection pattern. The network is composed of router chips, each with 4 *child* connections and either 2 or 4 *parent* connections. Each connection provides a link to another chip with a raw bandwidth of 20 megabytes/second in each direction. (Some of this bandwidth is devoted to addressing, tags, error checking, etc.) By selecting at each level of the tree whether 2 or 4 parent links are used, the bandwidths between nodes in the fat-tree can be adjusted. Flow control is provided on every link.

Based on technology, packaging, and cost considerations, the CM-5 bandwidths were chosen as follows. Each processor has 2 connections to the data network, corresponding to a raw bandwidth of 40 megabytes/second in and out of each processing node. In the first two levels, each router chip uses only 2 parent connections to the next higher level, yielding an aggregate bandwidth of 160 megabytes/second out of a subtree with 16 processing nodes. All router chips higher than the second level use all 4 parent connections, which, for example, yields an aggregate bandwidth of 10 gigabytes/second, in each direction, from one half of a 2K-node system to the other. The bandwidth continues to scale linearly up to 16,384 nodes, the largest machine that Thinking Machines can currently build. (The architecture itself scales to over one million nodes.) In larger machines, transmission-line techniques are used to pipeline bits across long wires, thereby overcoming the bandwidth limitation that would otherwise be imposed by wire latency. The machine is designed so that network bandwidth can be enhanced in future product revisions without affecting the architecture.

The network design provides many comparable paths for a message to take from a source processor to a destination processor. As it goes up the tree, a message may have several choices as to which parent connection to take. This decision is resolved by pseudorandomly selecting from among those links that are unobstructed by other messages. After the message has attained the height of the least common ancestor of the source and destination processors, it takes the single available path of links from that chip down to its destination. The pseudorandom choice at each level balances the load on the network and avoids undue congestion caused by pathological message sets. (Many naive algorithms for routing on mesh and hypercubic networks suffer from having specific message patterns that do not perform well, and the user is left to program around them.) The CM-5 data network routes all message sets nearly as well as the chosen bandwidths allow.

A consequence of the automatic load balancing within the data network is that users can program the network in a straightforward manner and obtain high performance. Moreover, an accurate estimate of the performance of routing a set of messages through the network can be predicted by using a relatively simple model [17]. One determines the load of messages passing through each arm of the fat-tree and divides this value by the available bandwidth. The

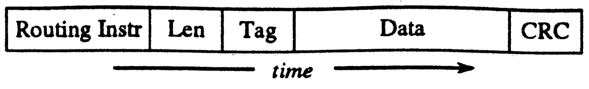

Figure 4: The format of messages in the data network. Each message contains routing instructions, a length field that indicates how many data words are in the message, a tag field that indexes an interrupt vector in the processor, data words, and a cyclic redundancy check.

worst-case such ratio, over all arms of the fat-tree, provides the estimate.

On random permutations, each processor can provide data into, and out of, the network at a rate in excess of 4 megabytes/second. When the communication pattern is more local, such as nearest neighbor within a regular or irregular two- or three-dimensional grid, bandwidths of 15 megabytes/second per processor are achievable. The network latency ranges between 3 and 7 microseconds, depending on the size of the machine. All of these empirical values include the time required for processors to execute the instructions needed to put messages into and take messages out of the network.

The data network is currently implemented from 1-micron standard-cell CMOS chips, with custom macro cells to provide high-performance circuits where needed. Each chip has an 8-bit-wide bidirectional link (4 bits of data in each direction) to each of its 4 child chips lower in the fat-tree, and 4 8-bit-wide bidirectional links to its parent chips higher in the fat-tree. The data-network chip can be viewed as a crossbar connecting the 8 input ports to the 8 output ports, but certain input/output connections are impossible due to the nature of the routing algorithm. For example, we never route a message from one parent port to another. When a message is blocked from its desired output port, it is buffered. Flow control information is passed in the reverse direction of message traffic to prevent buffer overflow. When multiple messages compete for the same output port, the arbitration is fair and prevents any link from being starved. We designed only one chip to do message routing, and we use the same chip for communication between chips on the same circuit board as between chips that are in different cabinets.

Interchip data is sent on differential pairs of wires, which increases the pin count of the chips, but which provides outstanding noise immunity and reduces overall power requirements. We rejected using separate transceivers at the packaging boundaries, because it would have increased power consumption, board real estate, and the number of different chips we would have needed to design, debug, test, stock, etc. The diagnostics can independently test each conductor of each differential signal, because differential signals are so immune to noise that they sometimes work even with broken wires.

The first 2 levels of the data network are routed through backplanes. The wires on higher levels are run through cables, which can be either 9 or 26 feet in length. The longer cables maintain multiple bits in transit. The wires in cables are coated with expanded Teflon, which has a very low dielectric constant. The cables reliably carry signals in excess of 90 percent of the speed of light.

The data network chips are clocked synchronously by a 40-megahertz clock. The clock is distributed with very low skew—even for the biggest machines—by locally generating individual clocks and adjusting their phases to be synchronous with a centrally broadcast clocking signal.

Messages routed by the data network are formatted as shown in Figure 4. The beginning of the message contains routing instructions that tell how high the message is to go in the tree and then the path it is to follow downward after it reaches its zenith. The routing instructions are chip-relative instructions that allow each chip to make a simple, local decision on how to route the message. Following the routing instructions is a field that indicates the length of the data in 32-bit words. Currently, the CM-5 network interface allows between 1 and 5 words. Longer messages must be broken into smaller pieces. Following the length field is a 4-bit tag field that can be used to distinguish among various kinds of messages in the system. The network interface interprets some of these tags as system messages, and the rest are available to the user. When a message arrives at a processor, the tag indexes a 16-bit mask register in the network interface, and if the corresponding mask bit is 1, the processor is interrupted. After the tag comes the data itself, and then a field that provides an integrity check of the message using a cyclic redundancy code (CRC).

Because we desired to build very large machines, we deemed it essential to monitor and verify the data network dynamically, because the chances of a component failure increase with the size of the system. Message integrity is checked on every link and through every switch. If a message is found to be corrupted, an error is signaled. Messages snake their way through the switches in a manner similar to cut-through [13] or worm-hole [3, 4] routing, and so by the time that a data-network chip has detected an error, the head of the message may have traveled far away. To avoid an avalanche of errors, the complement of a proper CRC is appended to the message. Any chip that discovers the complement of a proper CRC signals a secondary error. Thus, a typical error causes one chip to signal a primary error with a trail of chips reporting secondary errors, although there is some positive probability that a primary error is reported as a secondary error. Diagnostic programs can easily isolate the faulty chip or link based on this information, which is accessible through the diagnostic network. Lost and replicated messages can be detected by counters on each chip and in the network interfaces that maintain the number of messages that pass on each link. Using a variation on Kirchoff's current law, the number of messages entering any region of the network, including the entire network or a single chip, must eventually equal the number of messages leaving the region. This condition is checked for the entire data network by the control network (see Section 5).

Once a faulty processor node, network chip, or interconnection link has been identified, the fault is mapped out of the system and quarantined. The network interface allows for mapping faulty processing nodes out of the network address space. The rest of the system ignores all signals from the mapped-out portion, thereby allowing the system to remain functional while servicing and testing, or even powering down, the mapped-out portion.

When a chip or link in the data network fails, there are two mechanisms to map around the fault. Either the network can be configured to route messages away from the failure, or processing nodes that might use the chip or link can be mapped out. By picking the better of the two alternatives, the system can guarantee either that at most 6 percent of the network is lost or that at most 1/64 of the processing nodes are mapped out.

The network has a contract with processors that guarantees all messages are delivered. The contract says, *"The data network promises to eventually accept and deliver all messages injected into the network by the processors as long as the processors promise to eventually eject all messages from the network when they are delivered to the processors."* The data network is acyclic from inputs to outputs, which precludes deadlock from occurring if this contract is obeyed. To send a message, a processor writes the destination processor address and data to be sent to a memory-mapped outgoing FIFO in its network interface. The processor then checks whether the message was accepted by the network. If not, which may occur because flow control information indicates that the network has not removed enough of a previous message from the outgoing FIFO, the processor can try again later. The processor may not block or spin when attempting to put a message into the network, however, because that would violate the contract. Instead, the processor must attempt to receive any messages that have arrived. In the current implementation, the processor is involved in all transactions with the network.

Although the simple contract above can implement the sending of data through the network in a deadlock-free manner, it is not strong enough to allow some communication protocols to be implemented straightforwardly. Consider, for example, the *fetch-deadlock problem*: each processor wishes to fetch a value from another processor, and the processors have finite buffer space. The message traffic for a protocol that solves this problem corresponds to a round trip in the network: a request from one processor to another, followed by a response from the other to the one. In this scenario, one processor may receive requests for data from many processors, but unfortunately, be unable to send responses because its outgoing FIFO to the data network is busy. The outgoing FIFO will eventually free, according to the contract, but only if the processor continues to accept delivery of messages from the network. With finite buffer space, however, there is a limit to how many requests it can handle. When it runs out of buffer space, the processor will be forced to refuse delivery, thereby breaking the contract, and deadlock may result.

With buffer space proportional to the number of processors in the system, it is possible to construct a "round-trip" protocol that solves the fetch-deadlock problem. The key idea is to program a reservation mechanism [14] that ensures that at most a bounded number of messages are outstanding between any two processors at any time. A processor X does not attempt to send a message to another processor Y until Y informs X that it has room to handle the message. This protocol, which has been implemented on some parallel computing systems, including the CM-5, requires a substantial software overhead for bookkeeping.

The CM-5, however, provides another way to solve the fetch-deadlock problem in a simple fashion requiring no bookkeeping and only constant buffer space. Each processor has 2 outgoing and 2 incoming FIFO's in its interface to the data network: a *left* port and a *right* port. The topology of the network is such that all links reachable from the left port are unreachable from the right port and vice versa. Thus, the data network is really two independent, interleaved networks. To implement the round-trip protocol, requests can be sent on the left side of the network, and responses returned on the right side. If a processor cannot send a response on the right side and his constant-size buffer is full, he stops receiving on the left side. Since any processor requesting data has a place to put it, however, the processors can satisfy the contract on the right side, and the responses will eventually clear out. Because the responses on the right side will eventually clear out, a processor can always eventually accept every request that arrives on the left side, and thus the processors satisfy the contract on the left side. Consequently, deadlock cannot occur.

In fact, deadlock cannot occur even if responses are sent on both sides of the data network, as long as requests are sent on one side only. The data network requires no more than two sides, even when there are many intermediate destinations, because such a communication pattern can be broken into a collection of round trips.

The CM-5 programming systems (Fortran90, C*, and *Lisp) never allow a user to deadlock, because they implement deadlock-free protocols for communication. Deadlock can occur, however, if a programmer chooses to program the individual processing nodes directly. All he need do is break the contract that the processing nodes have with the data network: he writes code that sends messages but never attempts to receive them. This danger may seem quite alarming, but it is no more alarming than the danger that a user writes an infinite loop. On the CM-5, the user can send and receive messages without executing a system call, as is required on many other systems. By giving the user direct access to the network, the user can in some circumstances obtain greater efficiency than he could obtain with the communication routines available in the standard system libraries. If he does deadlock himself, or write an infinite loop, he does not affect any other user.

Each user partition in the CM-5 system is capable of being run in either a batch or a timesharing mode. The requirement for timesharing raises the issue of what should be done with messages that are in transit in the routing network when a user's timeslice has expired and another user must be given access to the partition. The system cannot afford to wait until the user completes his communication, since the communication may not terminate for a very long time, and, in fact, it may never complete if the user has deadlocked himself.

We considered several solutions to the problem of swapping users. For example, we considered entering a special routine that would pull messages out of the router and discard them. This solution was considered too expensive, because the user would be constantly forced to checkpoint the computation so that the discarded messages could be reconstructed. Moreover, if the user fills the network with messages that are all addressed to the same processing node, then the time to empty the router would be proportional to the machine size, which was deemed unacceptably long.

This problem of swapping users is solved in the CM-5 by putting the data network into *all-fall-down* mode. Instead of trying to route messages to their destinations, when a data-network chip is in all-fall-down mode, each message is routed downward according to a fixed permutation that has been preprogrammed by the system and which ensures that all-fall-down messages are distributed evenly among the processing nodes. In the worst case, each node receives only a small number of misdirected messages, even if all messages were headed for the same destination processor. The all-fall-down messages can then be saved in memory with the user's state. When the user's task is resumed, the system resends them to their true destinations. Even if a timeshared user deadlocks, this context-switching mechanism precludes him from unduly affecting the other users who are sharing his partition.

During our design of the all-fall-down mechanism, the problem arose of how to set all chips of a partition into this mode. We considered engineering a mechanism in which all chips were put into all-fall-down mode simultaneously, but even so, we considered it a lot of detailed engineering work to guarantee that messages between chips when all-fall-down was initiated would be handled properly. Instead, we adopted a simple protocol to allow chips in the data network to be put into all-fall-down mode in any order. The basic idea is that all-fall-down messages are marked as such. When a chip sees an all-fall-down message, it routes it downward according to the preprogrammed permutation, even if the chip is not in all-fall-down mode and is routing other messages in a normal fashion. Thus, once a message starts falling, it keeps falling and is never interpreted by any chip as anything but an all-fall-down message.

In summary, the CM-5 data network provides fast point-to-point communication of data, but as importantly, it provides flexible solutions to system problems.

4 Synchronized MIMD

The CM-5 is a *synchronized MIMD* machine. Whereas the data network in the CM-5 is responsible for moving data efficiently between pairs of processors, the CM-5's control network provides an infrastructure for the coordination and synchronization of an entire set of processors. Much as a conventional microprocessor is divided into control and datapath [10, Chapter 5], we found that partitioning communication into a control network and a data network led to a simpler, more efficient design. This section discusses why we adopted a synchronized MIMD execution model for the CM-5.

A major design goal of the CM-5 was to support the *data-parallel* programming model [2, 11] efficiently. The basic idea of data parallelism is that processing large amounts of data usually implies that the same operations are performed on all elements of large sets of data. Consequently, these operations can be performed in parallel. For example, an operation might be specified for all pixels of digitized image, in which case it can be performed in parallel on each of the pixels. Data-parallel languages—such as Fortran90, C*, and *Lisp—allow the programmer to express such operations naturally. The programmer applies parallel operations to an entire *set* of data simultaneously, and the system efficiently multiplexes the computation onto the processing nodes of the machine.

Traditionally, the data-parallel model has been supported by so-called *single instruction stream, multiple data stream* (SIMD) parallel computers, such as our previous machine, the Connection Machine CM-2. SIMD machines typically have two networks. Besides a message-routing network, these machines employ a *broadcast* network over which a front-end processor distributes instructions to the individual processing nodes in the system. All processing nodes receive and execute the same instruction at the same time. Based on data in its memory, however, a processor may decline to execute an instruction and sit idle instead. In the CM-2, the broadcast network is embellished with an OR network, which can compute a logical OR of boolean values, one value per processing node, and distribute the result back to the processing nodes.

There are many advantages of using a SIMD architecture to execute data-parallel code. When a parallel operation is applied to a large set of data, each processor receives the same instructions and executes the same code, thereby causing the operation to be applied to each of the individual elements. Since SIMD machines are highly synchronized, it is also easy to coordinate processors to perform cooperative actions. Moreover, all processors are doing much the same thing, and thus the broadcasting of instructions saves the need to implement instruction-fetch units in all of the processing nodes.

SIMD machines are less efficient, however, when different processors wish to execute different sections of code. The machine must step through each section of code serially while processors not interested in the particular section of code being executed sit idle. This loss of efficiency limits the flexibility of SIMD machines.

In contrast with the SIMD machine organization is the *multiple instruction stream, multiple data stream* (MIMD) organization of a parallel computer. In a MIMD machine, each processor executes its own instruction stream, and thus there is no loss of efficiency when processors execute different code. Typically, processors in MIMD machines communicate among themselves using message-passing techniques [19, 20] or through shared memory [6, 7], but there is little or no architectural support for coordinating and synchronizing sets of processors. A programmer must synthesize aggregate operations himself, resulting in considerable code complexity and loss in performance. Thus, the greater flexibility of MIMD comes at a great cost.

In the CM-5, we abandoned the SIMD architecture of its predecessor, the CM-2, in favor of a MIMD execution model, but we salvaged SIMD's best attributes: the ability to share data among processors efficiently and the ability to quickly synchronize sets of processors. To support the sharing of data, the control network provides a fast broadcast mechanism. To support the synchronization of sets of processors, the control network provides fast "barrier synchronization." These two mechanisms allow data-parallel code to be executed efficiently on what is otherwise a MIMD machine. We now briefly discuss how each of these mechanisms supports the data-parallel programming model.

To execute a data-parallel program on the CM-5, the control processor broadcasts a section of the data-parallel program to the processing nodes, rather than broadcasting the entire instruction stream, as in a typical SIMD machine. The idea of distributing a single program to multiple processors has been dubbed "SPMD," for *single-program, multiple data* [5]. Unlike shared-memory machines, in which processors must individually fetch the program from a central memory, however, in the CM-5, the control processor broadcasts the program to the processing nodes over the control network, and then the processors execute the program locally.

As long as a processor in a data-parallel programming environment does not communicate with other processors, it can execute code without worrying where in the code the other processors are. When processors communicate, however, program correctness often demands that processors know when it is safe to proceed to the code after the communication step. In particular, a processor may not know for a given communication pattern whether it will receive zero, one, or more messages, and thus it cannot determine whether it can proceed without some knowledge of whether other processors still have messages to send it. Consequently, the CM-5 provides a synchronization mechanism to inform all processors of the termination of message routing on the data network.

The CM-5 provides *barrier synchronization* (see, for example, [12, 21, 5]) via its control network. In barrier synchronization, a point in the code is designated as a barrier. No processor is allowed to cross the barrier until all processors have reached the barrier. In addition, the barrier mechanism in the CM-5 can check whether message routing is complete in the data network. By providing barrier synchronization in hardware, we avoided the complicated protocols that users often implement by hand on MIMD machines that are not synchronized. Since our mechanism is a parallel one, we also avoid the performance problems endemic to machines that support barriers through the use of shared semaphores.

We discovered four implementation advantages of synchronized MIMD over SIMD. First, the bandwidth in and out of a processing node is a critical resource. A program is typically much shorter than the instruction stream it generates. By broadcasting a program to the processing nodes, rather than sending its entire instruction stream, less of the bandwidth into a node is required for instructions, and hence more is available for communicating the user's data. Second, since processing nodes fetch their instructions locally, we were able to build the CM-5 from standard microprocessors rather than having to design our own. At the time of this decision, high-performance RISC microprocessors were just becoming available. We decided they were a good technology curve to "ride" and would allow us to focus more of our internal effort on networks and vector units, the bread and butter of high-performance computing. Third, the implementation of a control network gave us a platform to solve other system coordination problems. For example, if a user hangs up one or all of his processors, the operating system can broadcast a message that causes the processors to trap to supervisor code. Fourth, our synchronized MIMD architecture can execute more traditional MIMD code. For example, we have been able to port message-passing applications from other MIMD machines, and in many cases, to simplify and speed them up considerably by replacing their elaborate protocols with simple uses of our control network.

To summarize, the synchronized MIMD architecture of the CM-5 simply and efficiently provides the flexibility of MIMD and the SIMD ability to coordinate sets of processors.

5 The CM-5 Control Network

There are two general classes of operations on the control network: broadcasting and combining. Separate FIFO's in the network interface correspond to each type of control-network function. A processor pushes a message into one of the outgoing FIFO's, and shortly after all processors have pushed messages, the result becomes available to all processors as messages in their respective incoming FIFO's.

Every operation on the control network potentially involves every processing node. Broadcast messages from the control processor are replicated at nodes in the tree and distributed to the subtrees. Other operations, such as scans (parallel prefix), require input from all processors and provide output to all processors. The control network is pipelined, so that several messages can be sent before any are received. To provide further flexibility, each processing node can set up the network interface to abstain from certain control-network operations. These operations complete as if the abstaining processors had provided "identity" data, but without making them waste processing cycles. Overall, the control network is designed to support cooperative functions that require little bisection bandwidth, and hence, which can be implemented efficiently on a simple tree.

Broadcasting

A processor may broadcast a message through the control network to all other processors in its partition. The control network supports four kinds of broadcasting: user broadcast, supervisor broadcast, interrupt broadcast, and utility broadcast. User and supervisor broadcasts are essentially identical, except that supervisor broadcasts are privileged operations. These broadcast operations can be used to download code and to distribute data. An interrupt broadcast is a privileged operation that causes every processor to receive an interrupt. Interrupt broadcasts provide the ability to "grab the attention" of all processors in the user partition, which is especially useful for implementing operating system functions, such as swapping timeshared users. The utility broadcast is used by the operating system to configure partitions and to perform other sorts of system operations.

Only one processor may broadcast at a time, but broadcasts are pipelined so that the broadcasting processor can fully utilize the broadcast bandwidth of the network. If, while one processor is broadcasting, another processor sends a broadcast message, the control network signals an error when the competing messages collide. The number of simultaneous pipelined broadcasts supported by the control network depends upon the height of the network partition. The current implementation of the CM-5 provides the user with up to 8 words in a broadcast and the supervisor with up to 4 words.

Combining

The control network supports four different types of combining operations: reduction, forward scan (parallel prefix), backward scan (parallel suffix), and router done. Moreover, the network interface chip is capable of masking out processors that do not wish to participate in a control-network operation, so that operations can be performed only on a subset of the processors in a partition. Only one combining operation can be initiated at a time, but the network is pipelined, which allows several operations to be initiated rapidly in sequence.

A reduction operation combines values provided by all (participating) processors according to a user-supplied operator and delivers a copy of the result to all processors. Messages are combined with one of five operators on 32-bit words: bitwise logical OR, bitwise logical XOR, signed maximum (which also works for IEEE floating-point numbers), signed addition, and unsigned addition. (The two addition operators differ in how overflow is reported.) Reductions over other commonly occurring operators (such as bitwise logical AND) can be easily synthesized from these and local processor operations. The control network also supports reductions on values larger than 32 bits by a sequence of 32-bit reductions, each of which saves residual data, such as a carry in the case of addition, which is input to the next reduction in the sequence. Since the control network is pipelined, the latency for a multiple-word reduction operation is not unduly affected.

A forward scan operation delivers to the ith processor the result of applying one of the five reduction operators to the values in the preceding $i-1$ processors (in the linear order given by data network address). For example, a forward scan of the vector $\langle 3,2,0,4,2,6,5,8\rangle$ with the operator $+$ yields the vector $\langle 0,3,5,5,9,11,17,22\rangle$. A backward scan provides similar functionality in the reverse direction. Scans can be segmented: if a "segment start" bit in the network interface is set, the scan starts over at that point. Backward scans are also supported. All basic scan operations use 1-word (32-bit) inputs, but multiple-word scans are supported by a sequence of 1-word scans in a manner similar to multiple-word reductions. An excellent discussion of scans can be found in [2].

Early on in the design of the CM-5, we decided to support scans in hardware. Our experience with the CM-2 showed that many high-performance data-parallel algorithms—including both combinatorial and numerical algorithms—make extensive use of scans. The operations that were selected (OR, XOR, etc.) reflect a compromise between making the hardware fast and simple and providing sufficient building blocks out of which other operations could be constructed. For instance, OR can be used to implement AND (DeMorgan's law), so there is no need to implement both. As a more sophisticated example, segmented reductions, which are not provided directly by the hardware, can be implemented by using two segmented scans, one forward and one backward. Since the control network is pipelined, the overhead of doing both is minimal.

The router-done operation is a specialized reduction that lets the processors know when communications involving the data network are complete. In the data-parallel programming model, this operation is often required so that processors know when it is safe to proceed to the next data-parallel operation.

The basic idea behind the implementation of router-done is "Kirchoff's current law." When all processors have completed sending their messages and the number of messages that entered the data network equals the number that have left, the routing cycle is complete. The network interfaces keep track of the number of messages that enter and leave the data network. After a processor has completed sending all its messages, it pushes a message into the outgoing router-done FIFO. When all processors have sent messages into their outgoing FIFO's, the control network continually monitors the difference between the total number of messages put into the data network and the number removed from the data network. When this number becomes zero, each processor receives a message in its incoming router-done FIFO informing it that the data network is done routing messages. Using this "Kirchoff" method has the additional benefit that if a hardware error causes messages to be lost or created, the error can be detected and signaled, either by a failure of the router-done operation to complete on the one hand or by the unexpected arrival of a message after the router-done operation has completed on the other.

The CM-5 control network also supports one synchronous OR operation and two identical asynchronous OR operations that can operate in parallel with other network oprations, and have separate FIFO's in the network interface. The synchronous OR is similar to an OR reduction, except that a processor's input and output each consist of only a single bit. Each asynchronous OR operates continuously without waiting for all processors to participate. Processors are free to change their inputs at any time and sample the output. The asynchronous OR can be used for signaling conditions and exceptions. The transition of an asynchronous OR from 0 to 1 can be used to signal an interrupt. One of the two asynchronous OR's is privileged, and the other is nonprivileged.

The synchronous OR or any of the various combining operations can be used to implement ***split-phase*** barrier synchronization [23]. (In independent work [9], this type of synchronization has been called a *fuzzy barrier*.) In a split-phase barrier, the barrier is a region of code with an entry and an exit. (If the region is empty, an ordinary barrier results.) When a processor enters the split-phase barrier, it pushes an input message into an appropriate outgoing FIFO. Shortly after all other processors have pushed their messages, they all receive messages from the corresponding incoming FIFO, and each can infer that all have entered the barrier. The advantage of a split-phase barrier over an ordinary barrier is that the processor can execute code while waiting for the barrier to complete. Thus, just as the instruction following a delayed branch in a RISC architecture can compensate for the latency of the branch, the code between barrier entry and exit can compensate for the latency of synchronization. The router-done operation couples barrier synchronization with the test of whether routing on the data network has completed, so that no processor abandons its effort to receive messages until all processors are done sending them.

The control network also detects certain kinds of communication errors and distributes them throughout the system. For example, if two processors attempt to perform different combining operations, an error is signaled.

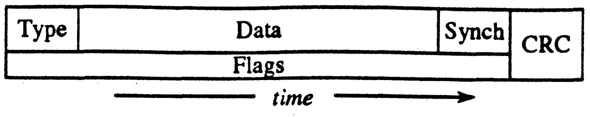

Figure 5: The format of messages in the control network. Each message contains a field that indicates the type of message, a 32-bit word of data, some synchronization bits, and various other flags. The message is checked using a cyclic redundancy code.

More importantly, hard errors detected by the data network and the network interfaces are collected by the control network. These error signals are combined using a logical OR and are redistributed to all the processors so that the operating system can isolate them and recover if possible.

Organization of the control network

The architecture of the control network is that of a complete binary tree with processing nodes, control processors, and I/O channels at the leaves. When a CM-5 system is configured, each user partition is assigned to a subtree of the network. Processing nodes are located at the leaves of the subtree, and a control processor is mapped into the partition as an additional leaf.

The control network is implemented using a 1-micron CMOS standard-cell chip that contains custom macro cells to implement high-performance circuitry. Like the data network chip, it uses a 40-megahertz clock. Three binary-tree nodes are packaged on each chip. There are 4 11-bit-wide bidirectional links (6 bits in the up direction and 5 bits in the down direction) to 4 child chips lower in the tree and 1 11-bit-wide bidirectional link to a parent. As in the data network, interchip signals are sent on differential pairs of wires.

Unlike data network packets, control network packets have a fixed length of 65 bits. (There is actually, in addition, a 5-bit packet used during system initialization to align the 65-bit packet boundaries so that a node can process the same fields in arriving messages at the same time.) The general format is illustrated in Figure 5. It is broken into two parallel streams, a major stream and a minor stream. The minor stream contains a variety of control bits, including various error and status flags, several flow-control bits, and a bit to implement segmented scans. The major stream begins with a packet description field, which defines the packet type—*single-source*, *multiple-source*, *idle*, or *abstain*—as well as the specific operation—user broadcast, supervisor broadcast, interrupt, scan (including combiner), reduce, etc. Then comes a 32-bit word of data. The major stream ends with a field containing the global synchronization bits. The entire packet is checked using a cyclic redundancy code (CRC), which is the last information in the packet to be transmitted.

The four packet types are processed differently by the control network. Whereas single-source packets are used to implement broadcasting, scans and reductions employ multiple-source packets. Idle packets are used as "filler" and are sent when a control network node has nothing better to ship. The abstain packet allows a control network node to proceed when it would otherwise wait for a multiple-source packet.

When a processor initiates a broadcast or interrupt through the control network, its network interface inserts a single-source message into the tree at a leaf. This message proceeds up to the root node of the user's tree, where it is turned around and distributed to all the processors in the partition. An error is signaled if two single-source packets from different sources meet at a control network node. If it meets with other kinds of packets, a single-source packet has priority. There is no buffering for single-source packets. Flow control for single-source packets is implemented by the network interface on an end-to-end basis.

Processing multiple-source packets is more involved. When a processor initiates a cooperative operation such as a scan, the network interface inserts a multiple-source message into the tree. At each internal node, a multiple-source message waits until its sibling's message has arrived. While a message is waiting, the node sends idle messages up the tree. When the sibling's message arrives, arithmetic or logical operations combine the two messages into one, which is sent up the tree. To implement scans, the message or its sibling may be put aside in another buffer to combine later with a value coming from the node's parent. When a multiple-source message finally reaches the root, it is sent downward. As it reencounters the internal nodes of the tree, it is replicated or further combined with waiting messages. (A good overview of the implementation of scans can be found in [2].)

While a multiple-source packet is waiting for a sibling or a parent, other packets arriving on the same input can be processed. If the newly arriving packet is a single-source packet, it proceeds ahead of the waiting packet, thereby giving priority, for example, to supervisor broadcasts and interrupts. If the new packet is another multiple-source packet, it is queued in the buffer behind the packets already waiting. Multiple-source packets thus maintain a consistent order, which allows two or more combining operations on the control network to be pipelined properly. Flow control in the network precludes buffers from overflowing.

An important requirement of the control network was that it be able to connect a control processor to each user partition. The control processor executes the scalar part of the data-parallel code, while the processing nodes execute the parallel part. We considered having scalar code executed by one or all of the processing nodes, but eventually decided that having a control processor associated with each partition would simplify matters. First, since the system cost of the control processor is very low compared with the multitude of processing nodes, we can afford to run it with large amounts of memory and with additional architectural features to enhance its performance. Consequently, the control processor is able to more efficiently execute scalar code than can a processing node. Second, the data-parallel code that runs on the earlier CM-2 machine is already split into scalar and parallel parts. Porting this code to the CM-5 was easier, since we could maintain the same split. Finally, since the control processor has a connection to an Ethernet, the user partition can run a standard Unix which communicates across the attached Ethernet.

At the end of a user's timeslice during timesharing, the control network can be flushed in a manner similar to a broadcast operation, aborting any user-level control-network operations in progress. The network interfaces retain the values that the user has pushed into the control network until the corresponding operation has completed, however. These values are saved as part of the user's state. When the user's task is resumed, the saved values can be used to reinitiate the control-network operations.

In case of a fault in a CM-5 processing node, network chip, or interconnection link, the control network—like the data network—can be configured to map the fault out of the system. The diagnostic network (see Section 6) can set internal switches within the control network to map out parts of the control network. Since the computations performed by the control network depend only on the control network being a binary tree, and not on its being a *complete* binary tree, computations within the control network can safely ignore the mapped-out portions of the system.

In addition, the control network has some additional switching capability to map around faults in the control network itself and to be able to connect any of the control processors to any partition. This additional switching capability is implemented as follows. Conceptually, each switch of the control network has 2 parents and 4 children and contains two binary-tree nodes which can be statically configured so that either can connect to any pair of children. By connecting these chips in a manner similar to the data network fat-tree, any control processor can be connected to any partition, subject to the availability of bandwidth. For example, if there are only 4 control network channels into a subtree, one cannot connect 5 control processors to 5 partitions in the subtree. Short of this bandwidth restriction, however, any connection of control processors to legal partitions can be implemented using an off-line routing algorithm similar to that in [16, Theorem 1].

In summary, the CM-5 control network provides the mechanisms to allow data-parallel code to be executed efficiently, as well as allowing more general kinds of parallel models to be implemented. Its structure as a binary tree provides an inexpensive way to provide the advantages of both traditional SIMD and traditional MIMD architectures.

6 The CM-5 Diagnostic Network

During the design of the CM-5, great emphasis was placed on system availability. Despite conservative design techniques and the use of proven circuit and interconnect technologies, the sheer size of the largest CM-5 systems forced us to abandon any attempt to achieve high availability by depending solely on inherent component reliability. Instead, our strategy relies on two architectural features of the machine: diagnosability which allows missing or broken hardware to be detected and isolated; and configurability, which allows most of the machine to operate when portions are broken or being serviced. This section shows how this strategy is implemented on the CM-5 through the use of a diagnostic network, the one network in the system that the user never sees.

One strategy to diagnose a parallel computer is to create diagnostic programs running on the processor nodes that exercise the processor nodes and various communications networks. When some part of the system fails to function correctly—for example, the data router fails to deliver a message or the control network produces the wrong answer for a combine operation, the diagnostic program itself may fail, because its correctness depends on the correct functioning of the system. We call such diagnostic programs *functionality dependent*. Our experience with the CM-1 and CM-2 exposed many of the limitations of functionality-dependent diagnostics. They are exceedingly difficult to write, they have nebulous coverage, and they lack precision in reporting the root cause of error conditions.

In contrast, diagnostics that are *functionality independent* rely on specific test structures, rather than the failure of normal system operation, to detect faults in the system. Using this kind of design-for-testability strategy, it becomes possible to view the CM-5 (or *any* sequential machine, for that matter) in terms of registers connected by combinational logic and wires. This change in perspective permits commercially available software tools to be used to generate high-coverage tests automatically for chips, boards, and the wiring that connects them. Moreover, when these tests fail, they provide specific information on the location and extent of the failure.

In the CM-5, design for testablity starts at the chip level. All CM-5 VLSI components support the IEEE 1149.1 testability architecture standard [1], also known as JTAG, for the Joint Test Action Group which originated the standard.[1] At the system level, the CM-5 diagnostic network provides parallel access to all system components from a *diagnostic processor*. The JTAG standard and the diagnostic network combine to form a diagnostic system which can quickly perform an in-system check of the integrity (over 99 percent single stuck-at fault coverage) of all CM-5 chips that support the JTAG standard and all networks.

Let us briefly review the JTAG interface standard. The JTAG standard provides a 4-pin interface for each chip in a system. On each chip, two pins provide input and output, respectively, for a selectable scan chain within the chip.[2] The standard specifies the *boundary scan register* (BSR) which connects all I/O pads in the chip into a bit-serial shift register. Two other pins serve as clock and control inputs. By scanning data in and out of chips, the BSR can be used to apply stimulus to the chip core for chip tests, or to monitor inputs and control outputs of the chip for connectivity tests.

In the CM-5, we extended the JTAG standard to include full internal scan in all proprietary chips. Details of this design are described in [25]. The use of a full internal scan allows software for automatically generating test patterns to generate a set of scan vectors with very high fault coverage. The vectors can be applied through the JTAG interface to test individual chips when they are manufactured and packaged. Later, when the chips are assembled into a system, the same tests can be applied through the diagnostic network.

The JTAG interface is designed to extend to multichip systems. When more than one chip is incorporated in a system, the scan paths are linked together in series by connecting the output from one scan path to the input of the next in a daisy-chain fashion. The clock and control pins are connected in parallel so that these signals can be broadcast to all chips in the chain.

Previous designs have focused on reducing the length of very long scan chains by placing scan-controllable bypass elements in the scan chain [22]. Unfortunately, testing all the chips in the system still requires serial access to each one. Even with ideally short test times on the order of seconds per device, this method would be unacceptably slow for an entire 16,384-node CM-5 comprising many tens of thousands of devices. Moreover, this method fails to take advantage of the inherent parallelism that can be achieved by testing large numbers of identical system components. For these reasons, it was evident early on in the design of the CM-5 that we needed a parallel strategy for supporting scan-based diagnostics.

The CM-5 diagnostic network provides simple and reliable access to the system components of the CM-5. It provides scan access to all chips supporting the JTAG standard, and programmable *ad hoc* access to non-JTAG chips. The diagnostic network itself is completely testable and diagnosable. The diagnostic network is able to map out and ignore parts of the machine that are faulty or powered down. It can be partitioned consistently with user partitions. The network is able to select and access groups of system chips in parallel, including:

- a single chip;
- a single type of chip;
- the chips within a user partition;
- the chips associated with a geographical portion of the system, *e.g.*, a given board, backplane, cabinet, etc.; and
- unions and intersections of previously specified sets of chips.

The diagnostic network is organized as a (not necessarily complete) binary tree, at the root of which sit one or more diagnostic processors, and at the leaves of which are *pods*. Each pod is a physical subsystem, such as a board, which directly supports the JTAG interface. At any given time, a single diagnostic processor controls the diagnostic network. From the root of the tree, an individual pod can be addressed by giving a binary number, each bit of which corresponds to a level in the tree and specifies a path from the root to the leaf: bit i of the address specifies whether the addressed leaf is in the left or right subtree of the node at level i. If the height of the tree is h, then h bits are sufficient to specify any leaf.

[1] In the current implementation of the CM-5 architecture, neither the SPARC processor nodes nor the DRAM chips support the JTAG interface. Given the growing acceptance of JTAG standard, however, it is likely that off-the-shelf processors and memory will support the standard in the near future. The CM-5 architecture is designed to incorporate these JTAG-supporting chips when they become available.

[2] This use of the term "scan" has nothing whatsoever to do with parallel prefix and suffix computations, as discussed in Section 5.

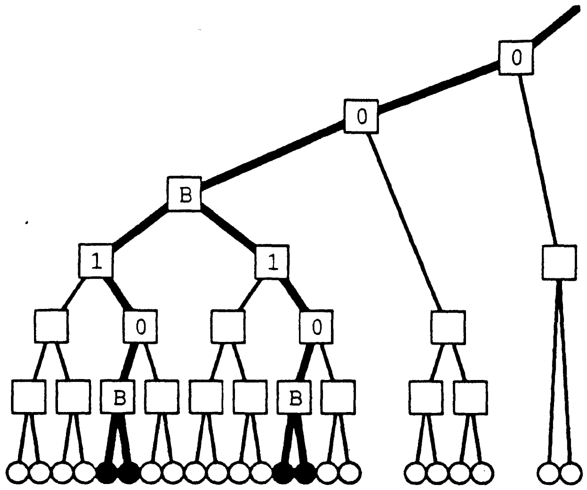

Figure 6: Steering a token down the diagnostic network. The address is decoded digit-serially, where each digit is 0, 1, or B, representing a selection of the left subtree, right subtree, or both subtrees, respectively. The example shows the selection made by the address 00B10B.

The diagnostic network allows groups of pods to be addressed according to a "hypercube address" scheme. For a tree of height h, a ***diagnostic virtual address*** is an h-digit number in which each digit is a 0, 1 or B. The B ("both") digit is a "wild-card" that matches both 0 and 1. For example, in a height-6 tree, the address 00B10B addresses the set $\{000100, 000101, 001100, 001101\}$, or $\{4, 5, 12, 13\}$. The addressing scheme can also be used to address the internal nodes of the diagnostic network by specifying addresses with fewer than h digits.

The decoding logic to implement the diagnostic virtual addressing scheme is based on the notion of steering "tokens" down the tree, as is illustrated in Figure 6. The mechanism works as follows. A token is inserted at the root of the tree together with a diagnostic virtual address, which is piped digit-serially into the root of the tree, high-order digit first. The root selects its right, its left, or both of its subtrees based on the high-order digit. If both subtrees are selected, the token splits into two tokens. Subsequent digits then steer the tokens and subsequent digits down the selected paths. When the end of the address is encountered, the nodes holding tokens are considered to be selected, and nodes on paths from them to the root provide the conduit for control.

Tokens and their paths from the root stay in place until a subsequent address erases them or until they are explicitly erased. This feature can be employed to combine two sets of selected nodes. For conceptual simplicity, suppose each of the two sets of nodes is in a separate subtree of the root. First, the left set is selected using a 0 as the high-order digit and pushing a token down the appropriate paths. Next, the right set is selected using a 1 as the high-order digit and pushing a token down the appropriate paths. The left set remains intact, but is temporarily inaccessible from the root because the right set is being selected. Finally, we push another token with an address of B to select the root itself and cause it to enable both its children, thereby merging the two sets. More complicated set unions are possible using this basic mechanism.

Most of this mechanism is hidden from the diagnostic engineer. Software extends the diagnostic virtual address within pods to address individual chips. Software also converts between the diagnostic network addresses and two other kinds of addresses: ***geographical addresses***, which specify cabinets, backplane, slot type, slots, etc.; and ***network addresses***, which give the locations of components according to the data and control networks' view of the machine. In general, important subsets of geographical addresses can be specified with one diagnostic virtual address. Important subsets of network addresses—for example, all data network chips at a given height in the machine, or all boards containing processing nodes in some contiguous range—typically take a combination of at most h diagnostic virtual addresses, where h is the number of bits in the address. The most important aspect of the addressing scheme, however, is that the time to access the various subsets does not grow by more than a small additive amount when the size of the machine doubles.

Having addressed a subset of the pods in the system, scan vectors can be applied in parallel to detect errors. JTAG serial data and control inputs are broadcast to all selected pods. Each pod provides a scan output signal that can be OR'ed or AND'ed with the corresponding signals from the other selected pods. The choice of an OR or AND combiner depends on what the diagnostic processor is expecting for a scan result. If the expected bit is a 1, the AND combiner is chosen. The result of the combining is a 1 if and only if all selected pods assert a 1. Similarly, if the expected output is a 0, the OR combiner is chosen. The result of the combining is a 0 if and only if all selected pods assert a 0. If an error is detected in a group of selected pods, the offending pod can be isolated either by addressing each pod in the group individually one at a time, or by a divide-and-conquer methodology. Within a pod, standard techniques for finding errors within a serial chain of JTAG interfaces are used to isolate the error to the chip level.

Since the diagnostic network is a tree, it is relatively easy to make it self diagnosing. Each level beneath the root can be tested by the levels above. Moreover, since there is not much logic in the diagnostic network, the probability that the network fails itself is much less than the probability that other parts of the system fail. Moreover, since the network is a tree, most of its logic is near the leaves, so that when a part of the diagnostic network does fail, only a small part of the tree is likely to be isolated. We did not mind relying on relatively few components near the root, since any small set of components is quite reliable—it is only large aggregates which have a high probability of failing.

The current implementation of the diagnostic network uses essentially two off-the-shelf chips. The address decoding of a binary node is implemented with a P22V10 24-pin PAL, and the finite-state control of a node is implemented with a P18V8 20-pin PAL. The chips can be clocked at any speed up to about 1 megahertz. In some places in the system, to save chips, address decoding of a 4-ary or 8-ary node is implemented directly as a single-chip PAL, rather than by using several separate binary-node PAL's.

7 Conclusion

We conclude this paper with a brief history of our implementation effort.

Work on the CM-5 architecture was begun in the latter part of 1987. We performed network simulations that led us, by January 1988, to choose a fat-tree architecture for the data network. By May 1988, most of the data network logic had been designed and verified, although several changes were implemented during the summer of 1988. A register-transfer-level (RTL) description of the data network chip was completed in early 1989, and the data network architecture was frozen. A gate-level description of the data network chip was completed by the early summer of 1989. The JTAG diagnostic interface was debugged using the data network chip design as a framework. The data network chip also served as the guinea pig for system and chip timing software. The chip was submitted for fabrication in May 1990.

The MIMD-plus-control-network design was proposed in early 1988, but we did not officially decide to use it until May 1989. Until then, we maintained other potential design alternatives. Work on the control network chip and the network interface proceeded concurrently. By the end of summer 1989, RTL models of both were simulating successfully. Gate-level models were implemented by the end of December 1989, and the control network architecture and network interface were frozen shortly thereafter. In May 1990, both the control network chip and the interface chip were submitted for fabrication.

The strategy of the diagnostic network was laid out in 1988, but work did not begin on it in earnest until the fall of 1989. Most of the work involved implementing the JTAG interface on the various chips. The design of the diagnostic network itself took only a few months, but considerable effort in 1990 and 1991 went into diagnostic software.

In the latter part of 1990, our attention turned to system integration. We received and tested the data network chips in July 1990, the control network chips in August, and the interface chips in September. Within two days after the interface chips arrived, we had assembled the networks for a 2-node machine and powered it up, a feat due in large measure to our functional verification methodology [18]. That same day, the operating system—which had been developed on a simulator—functioned correctly on the machine. By year's end, we had successfully constructed several small machines, including a 64-node machine, some of which were dedicated to software development.

The year 1991 began with an effort to build a 256-node machine using a completely new mechanical design. Initially, it had been more important to make machines available to our software engineers than to construct a large machine. To test the limits of our physical design, however, we needed to build large machines. The 256-node machine was begun in February, and finished in March. The time frame was dominated by the build time in manufacturing. In May, we built a 544-node machine, which was shipped in August to the Minnesota Supercomputer Center on behalf of the Army High Performance Computer Research Center.

In October 1991, the Connection Machine Model CM-5 Supercomputer was publicly announced.

Acknowledgments

The CM-5 project was managed by Rolf-Dieter Fiebrich. Many others at Thinking Machines contributed to its success, but space does not permit us to thank them all individually. Of special mention, however, are Pam Chulada-

Smith, for keeping our systems running; Ken Crouch, for helping to design and test the data network protocols; Rick Epstein, who managed the hardware design of boards; Don Moodie, who engineered the mechanical design; Sami Nuwayser, for his system administration and chip-design talents; Dave Patterson, for encouraging us to resurrect our MIMD-plus-control-network design; David Potter, who led the electrical and packaging effort; Linda Simko, whose administrative talents saved us from needing to have any; Cindy Spiller, for writing the initial system code and helping to verify the system; Guy Steele, for insights into the linguistic implications of the network architecture, and Jon Wade, who contributed to the design of the differential pads and system clocking.

In addition, we would like to thank Keira Bromberg, Dan Cassiday, Chung Der, John Devine, John Earls, Jamie Frankel, Steve Goldhaber, Harold Hubschman, Jim Lalone, Don Newman, Ralph Palmer, Gary Sabot, Soroush Shakibnia, Craig Stanfill, John Sullivan, Lew Tucker, Bob Vecchioni, Charlie Vogt and Bruce Walker for their contributions. We also thank the representatives from our chip vendors—Mark O'Brien, Curt Dicke, Laura Le Blanc, Jerome Li, and Shabnam Zarrinkhameh of LSI Logic; and Jerry Frenkil, Frank Kelly, Marc Corbacho, Paul Pinelle, and George Wall, of VLSI Technology, Inc.—for their help in producing the network chips.

References

[1] IEEE Std 1149.1-1990. IEEE standard test access port and boundary-scan architecture, 1990.

[2] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[3] W. J. Dally. Wire-efficient VLSI multiprocessor communication networks. In Paul Losleben, editor, *Proceedings of the 1987 Stanford Conference on Advanced Research in VLSI*, pages 391–415, Cambridge, MA, 1987. The MIT Press.

[4] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocesor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[5] F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. Research Report RC 11552, Computer Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 1986.

[6] M. Dubois and S. Thakkar, editors. *Cache Architectures in Tightly Coupled Multiprocessors.* IEEE Computer Society, June 1990. Special Issue of *Computer*, Volume 23, Number 6.

[7] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer — designing a MIMD, shared-memory parallel machine. *IEEE Transactions on Computers*, C-32(2):175–159, February 1983.

[8] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. *Advances in Computing Research*, 5:345–374, 1989.

[9] R. Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Boston, Massachusetts, 1989.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, 1990.

[11] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[12] H. F. Jordan. A multi-microprocessor system for finite element structural analysis. In A. K. Noor and Jr. McComb, H. G., editors, *Trends in Computerized Structural Analysis and Synthesis*, pages 21–29. Pergamon Press Ltd, 1978. Published as a special issue of *Computers & Structures*, Volume 10, Numbers 1–2.

[13] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[14] Leonard Kleinrock. Principles and lessons in packet communications. *Proceedings of the IEEE*, 66(11):1320–1329, November 1978.

[15] F. T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *29th Annual IEEE Symposium on Foundations of Computer Science*, pages 256–271, 1988.

[16] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[17] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[18] M. St. Pierre, S.-W. Yang, and D. Cassiday. Functional VLSI design verification methodology for the CM-5 massively parallel supercomputer. In *International Conference on Computer Design*, October 1992. To appear.

[19] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–23, January 1985.

[20] C. L. Seitz, W. C. Athas, W. J. Dally, R. Faucette, A. J. Martin, S. Mattisson, C. S. Steele, and W.-K. Su. *Message-Passing Concurrent Computers: Their Architecture and Programming.* Addison-Wesley, Reading, MA, 1986.

[21] P. Tang and P.-C. Yew. Processor self-scheduling for multiple-nested parallel loops. In K. Hwang, S. M. Jacobs, and E. E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[22] Texas Instruments. *SN54ACT8997, SN74ACT8997 Scan Path Linker With 4-bit Identification Bus*, April 1990. Product Preview.

[23] Thinking Machines Corporation, applicant. W. Daniel Hillis, inventor. European Patent Application Serial Number 89 902 461.6, priority date of February 2, 1988, entitled *Method and Apparatus For Aligning The Operation Of A Plurality Of Processors.* Also International Application Number WO 89/07299 (Published under the Patent Cooperation Treaty), Publication Date 10 August 1989.

[24] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.

[25] R. Zak and J. Hill. An IEEE 1149.1 compliant testability architecture with internal scan. In *International Conference on Computer Design*, October 1992. To appear.