PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

A Universal Parallel Computer Architecture

William J. Dally

Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
billd@ai.mit.edu

Abstract

Advances in interconnection network performance and interprocessor interaction mechanisms enable the construction of fine-grain parallel computers in which the nodes are physically small and have a small amount of memory. This class of machines has a much higher ratio of processor to memory area and hence provides greater processor throughput and memory bandwidth per unit cost relative to conventional memory-dominated machines. This paper describes the technology and architecture trends motivating fine-grain architecture and the enabling technologies of high-performance interconnection networks and low-overhead interaction mechanisms. We conclude with a discussion of our experiences with the J-Machine, a prototype fine-grain concurrent computer.

1 Introduction

Computer architecture involves balancing the capabilities of components (processors, memories, and communication facilities), organizing the connections between the components, and choosing the mechanisms that control how components interact. The top-level organization of most computer systems is similar. As shown in Figure 1, all parallel computers consist of a set of processing nodes each of which contains a processor, some memory, and a communication interface. The nodes are interconnected by a communication facility (typically a network). A sequential processor is the special case where there is only a single node and the network is used only to connect to I/O devices.

At present, the organization of processors and memories is well understood and network technology is rapidly maturing. While these components continue to evolve with improving technology and incremental architecture improvements, they do not provide significant differentiation between machines. With a convergence in machine organization, balance and mechanisms become central architectural issues and serve as the major points of differentiation.

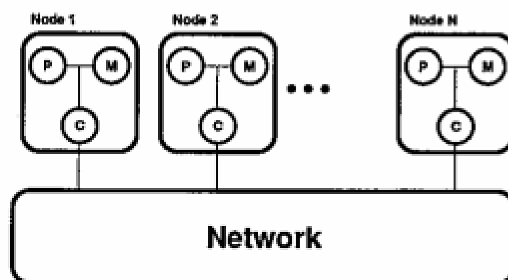

Figure 1: The structure of a parallel computer or multicomputer. All multicomputers consist of a collection of nodes connected by a network. Each node contains a processor (P), a memory (M), and a communication interface (C). Machines differ in the balance of component performance and in the mechanisms used for communication and synchronization between the nodes.

This paper explores two ideas related to balance and mechanisms. First, we propose balancing machines by cost, rather than by capacity to speed ratios. Such cost-balanced machines have a much higher ratio of processor to memory area and hence much greater processor throughput and memory bandwidth per unit cost compared to conventional machines. Cost-balanced machines are have a fine-grained physical structure. Each node is physically small and has a small amount of memory. Efficient operation with this fine-grained structure depends on high-performance communication between nodes and low overhead interaction mechanisms.

The mechanisms that control the interaction between the nodes of a parallel computer determine both the

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation, and in part by assistance from Intel Corporation.

grain-size and the programming models that can be efficiently supported. By choosing a simple, yet complete, set of primitive mechanisms, a parallel computer can support a broad range of programming models and operate at a fine grain size.

A fine-grain parallel computer with fast networks and efficient mechanisms has the potential to become a universal computer architecture in two respects. First, this class of machine has the potential to universally displace conventional (sequential and parallel) coarse-grained computers. Secondly, a simple yet efficient set of interaction mechanisms serves as the basis for a parallel computer that is universal in the sense that it runs any parallel programming system.

The remainder of this paper explores the issues of balance and mechanisms in more detail. The next section identifies trends in conventional sequential processor architecture that have led to a cost-imbalance between processors and memory. Section 3 discusses how an opportunity exists to greatly improve the performance/cost of computer systems by correcting this imbalance. The next two sections deal with the two enabling technologies: Networks (Section 4) and Mechanisms (Section 5). Together these enable fine-grain machines to give sequential performance competitive with conventional machines while greatly outperforming them on parallel applications. Our experience in building and operating a prototype fine-grain computer is described in Section 6.

2 Trends in Sequential Architecture

Two trends are present in the architecture of conventional computers:

1. The size of a processor relative to the size of its memory system is decreasing exponentially.
2. The time required for a processor to interact with an external device connected to its memory bus is increasing.

The first trend is due to an attempt to balance computer systems by ratio of processor performance (i/s) to memory capacity (bits). In 1967, Amdahl [22] suggested that a system should have 8Mbits of memory for each Mi/s of processor performance. The processor performance/size ratio (i/s $\times$ cm^2) benefits from technology improvements in both density and speed while the memory capacity/size ratio (bits/cm^2) benefits only from density improvements. Thus the processor to memory cost ratio for an Amdahl-balanced system scales inversely with speed improvements.

Let $K(67)$ denote the ratio of processor cost to memory cost for such an Amdahl-balanced system in 1967. Every Y years, the line width of the underlying semiconductor technology has halved. As a result, the area of both the processor and the memory was reduced by a factor of four [23]. At the same time, the processor speed increased by a factor of α. To keep such a system Amdahl-balanced, the capacity (and hence the size) of the memory must also be increased by α. Thus, the processor to memory ratio during year $x > 67$ is given by $K(x) = K(67)\alpha^{(67-x)/Y}$. For typical values of $\alpha = 3$ and $Y = 5$ [23], $K(92) = .004K(67)$.

The cost of a conventional machine has become largely insensitive to processor size as a result of this exponential trend in the ratio of processor to memory size. Thus, processor designers have become lavish in their use of area[1]. Costly features such as large caches, complex data paths, and complex instruction-issue logic are added even though their marginal affect on processor performance (compared to a small cache and a simple organization) is minor. As long as the size of the machine is dominated by memory, adding area to the processor has a small effect on overall size and cost.

The second trend, the increase in external interaction latency, is due to the first trend, to the increasing difference in on-chip to off-chip signal energies, and to to deepening memory hierarchies. As processors get faster and memory size increases the number of processor cycles required to access memory increases. Modern microprocessor-based computers have a latency of 5-20 cycles for a main memory access and this number is increasing. At the same time, decreasing on-chip signal energies require greater amplification to drive off-chip signals. Also, as more levels of caching are introduced, the number of cycles expended before initiating an external memory reference increases and the memory interface becomes specialized for the transfer of cache lines.

If a conventional processor is used in a parallel computer, its high external interaction latency limits its communication performance as the network must typically be accessed via the external memory interface. Whether this interface uses DMA to transfer data stored in memory (and possibly cached) or uses writes to a memory-mapped network port, each word of the message must traverse the external memory bus and the cost of initiating an external memory operation is incurred at least once. The slow external memory interface also contributes to the lack of agility in modern processors (that is, their slowness in responding to external events and switching tasks) because a great deal of processor state must be transferred to and from memory during these operations.

These trends in conventional processor architecture

[1] As a result of this lavish use of area, processor sizes have scaled slightly slower than predicted by the formula above.

make conventional processors ill-suited for use in a parallel computer. Current cost-insensitive processors are not cost effective in a machine with higher processor to memory ratio where the cost of the processor is an important factor. Their high external interaction latency severely limits their communication performance and their poor agility limits their ability to handle synchronization.

This does not mean, however, that conventional instruction set architectures (ISAs) are unsuitable for parallel computing. Rather it is the cost-insensitive design style, deep memory hierarchies, and poor agility that are the problem. As we will see in Section 5, a conventional ISA can be extended with a few instructions to provide an efficient set of parallel mechanisms.

Most importantly, the trend toward ever higher memory to processor size ratios has created an enormous opportunity for parallel computing to improve the performance/cost of computers. By adding more processors while keeping the amount of memory constant, the performance of the machine is dramatically increased with little impact on cost. The current trend, however, of building parallel computers by simply replicating workstation-sized units (increasing processors and memory proportionally) does not exploit this advantage. The processor to memory ratio must be decreased to improve efficiency. This theme is explored in more detail in the next section.

3 Balance

Balance, in the context of computer architecture, refers to the ratios of throughput, latency, and capacity of different elements of a computer. In this section we will explore the balance between processor throughput, memory capacity, and network throughput in a parallel computer. A case will be made for balancing machines based on cost[2].

Traditionally, machines have been balanced by rules of thumb such as the one due to Amdahl discussed above. However, a more economical design results if a machine is balanced based on cost. A machine is *cost-balanced* when the incremental peformance increase due to an incremental increase in the cost of each component is equal. Let each component k_i in a machine with performance P have cost c_i, then the machine is *cost-balanced* if $\partial P/\partial c_i = \partial P/\partial c_j; \forall i, j$ [7].

It is difficult to solve these balance equations because (1) no analytic function exists that relates system performance to component cost and (2) this relationship varies greatly depending on the application being run. Also, analyzing existing applications can be misleading as they have been tuned to run on particular machines and hence reflect the balance of those machines.

A workable approach is to start from the present memory-dominated system and increase the processor and network costs until they reach some fraction of total cost, for example 10%. At this point the system costs a small fraction more than a conventional system. If designed with an appropriate communication network (Section 4) and mechanisms (Section 5), it should provide sequential performance comparable to that of a conventional machine. Applications that are parallelized to take advantage of the machine can potentially speed up by the entire increase in processing cost.

To make reasonable balancing decisions, it is important to use manufacturing cost, not component price, as our measure of cost. This avoids distorting our analysis due to the widely varying pricing policies of semiconductor vendors. To simplify our analysis of cost, we will use silicon area normalized to half a minimum line width, λ, as our measure of cost [27].

First consider the issue of processor to memory balance. There are two issues: (1) how large a processor to use on each node and (2) how much memory per processor. A 64-bit processor with floating point but no cache and simple issue logic currently costs about $100\text{M}\lambda^2$, about the same as 500Kbits of DRAM, and has a performance of 50Mi/s. Making a processor larger than this gives diminishing returns in performance as heroic efforts are made to exploit instruction-level parallelism [20]. A smaller processor may improve efficiency slightly. If we are allocating 10% of our cost to processors, we will build one processor for every 5Mbits of memory – rounding up this gives one processor per MByte. In today's technology a processor of this type with 1MByte of memory can easily be integrated on a single chip. In comparison, an Amdahl-balanced machine would provide 64MBytes of memory for each processor and be packaged in 30-50 chips.

Providing a small cache memory for the processor is cost effective; however a large cache and/or a secondary cache are not. Adding a small 4KByte I-cache and D-cache requires about $16\text{M}\lambda^2$ of area and greatly boosts processor performance achieving hit rates greater than 90% on many codes [35]. Making the cache much larger or deepening the memory hierarchy would greatly increase processor area with a very small return in performance. Also, using a small co-located memory reduces processor access time to DRAM memory.

The network to memory balance is achieved in a similar manner, by adding network capability until cost is increased by a small fraction. A great deal of network performance comes at very little cost. The PC (printed-circuit) boards on which the processor-memory chips are mounted have a certain wiring capacity and the periphery of the chips can support a certain number of I/O

[2]Much of the material in this section is based on a joint work in progress with Prof. Anant Agarwal of MIT.

pads[3]. The network can make use of most of these pin and wire resources at a very small cost. The cost of the network router itself is small; a competant router can be built in less than $10M\lambda^2$ [16]. For example, a router on an integrated processor-memory chip could easily support 6 16-bit wide channels from which a 3-D network can be constructed (Section 4). Conventional PC boards and connectors can easily handle these signals.

Attempting to increase network bandwidth beyond this level becomes very expensive. To add more channel pins, the router must be moved to a separate chip or even split across several chips incurring additional overhead for communication between the chips. These chips are pad-limited and most of their area is squandered. If the amount of memory per node is increased proportionally to the cost of the network router to hold the memory to network cost ratio constant, the network bandwidth per bit of memory decreases (and the processor to memory ratio is distorted).

A computer design can be approximately cost-balanced by using technology constraints to determine the processor/memory/network ratios. A simple three step method gives a well cost-balanced system:

1. Size the processor to the knee of its performance/cost curve to get a cost effective processor.
2. Set the processor to memory ratio to allocate a fixed fraction γ (in the example above 0.1) of cost to the processor to get a machine that is within $1/1-\gamma$ of the optimium cost.
3. Holding processor and memory sizes constant, size the network to the knee of its performance/cost curve to get a cost effective network.

Machines that are cost-balanced using this method offer aggregate processor performance and local memory bandwidth that is 50 times that of an Amdahl-balanced machine per unit cost. This performance advantage will expand by a factor of α every Y years.

Why are coarse-grained Amdahl-balanced machines widespread both in uniprocessors and parallel computers? In uniprocessors, the number of processors is not a free variable. Thus the designer is driven to increase the size and cost of a single processor far past the knee of its performance/cost curve.

Existing parallel computers are driven to a coarse grain-size because (1) they are built using processors that lack appropriate mechanisms for communication and synchronization, (2) their networks are too slow to provide fast access to all memory in the machine[2], and (3) converting software to run in parallel on these machines requires considerable effort [21]. Much of the difficultly associated with (3) is due to the partitioning requied to get good performance because of 1 and 2.

For cost-balanced machines to be competitive, increasing the number of processors must (1) not substantially reduce single-processor performance and (2) must provide the potential for near-linear speedup on certain problems. To retain single-processor performance on a machine with a small amount of memory per node, the network and processor communication mechanisms must provide a single processor access to any memory location in the machine in time competitive with a main memory access in a conventional machine. Single-processor performance depends on network latency. To provide speedup on parallel applications, the processor's communication and synchronization mechanisms must provide for low-overhead interaction and the network throughput must be sufficient to support the parallel communication demands. Parallel speedup depends on throughput and agility.

The two key technologies for building cost-balanced machines are efficient networks, and processor mechanisms for communication and synchronization. The next two sections explore these technologies in more detail.

4 Network Architecture and Design

The interconnection network is the key component of a parallel computer. The network accepts messages from each processing node of a parallel computer and delivers each message to any other processing node. Latency, T, and throughput, λ_s, characterize the performance of a network. Latency is the time (s) from when the first bit of the message leaves the sending node to when the last bit of the message arrives at the receiving node. Aggregate throughput λ_{sN} is rate of message delivery (bits/s) when the network is fully loaded.

T must be kept low to achieve good performance for sequential codes and for the portions of parallel codes where the parallelism is insufficient to keep the machine busy. During these periods performance is latency-limited and execution time is proportional to T. During periods where there is abundant parallelism, performance is throughput limited. Recent developments in network technology give throughputs and latencies that approach physical and information theoretic bounds given pin and wire constraints. A detailed discussion of this technology is beyond the scope of this paper. This section briefly summarizes the major results.

An interconnection network is characterized by its topology, routing, and flow control [11]. The topology of

[3]Typical PC boards support 20wires/cm on each of 4-8 wiring layers. Typical ICs support 100pads/cm along their periphery with 20-50% of these pads reserved for power.

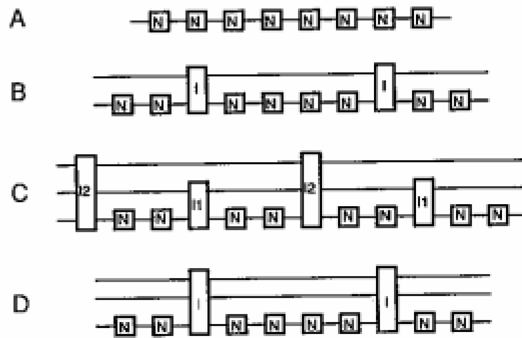

Figure 2: Insertion of express channels into a k-ary 3-cube gives performance within a small factor of physical limits : (A) One dimension of a regular k-ary 3-cube network, (B) Inserting one-level of express channels optimizes the ratio of wire to node delay for messages travelling long distances, (C) Hierarchical express channels also reduce the number of switching decisions to the minimum, $\log_q N$. (D) Adding multiple channels at each level adjusts network bisection bandwidth to maximize throughput.

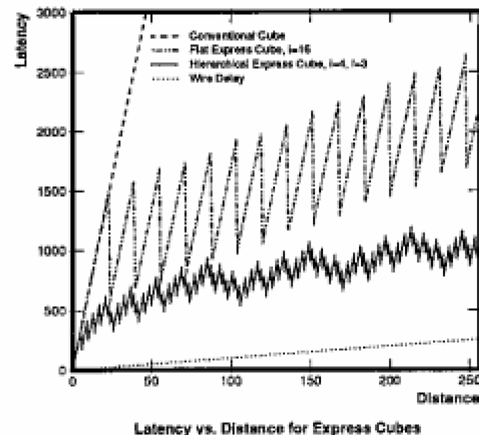

Figure 3: Latency as a function of distance for a hierarchical express channel cube with $i = 4$, $l = 3$, $\alpha = 64$, and a flat express channel cube with $i = 16$, $\alpha = 64$. In a hierarchical express channel cube latency is logarithmic for short distances and linear for long distances. The crossover occurs between $D = \alpha$ and $D = i\alpha \log_i \alpha$. The flat cube has linear delay dominated by T_n for short distances and by T_w for long distances.

a network is the arrangement of nodes and channels into a graph. Routing specifies how a packet chooses a path in this graph. Flow control deals with the allocation of channel and buffer resources to a packet as it traverses this path.

The topology strongly affects T since it determines (1) how many hops H a message must make, (2) the total wire distance D (cm) that must be traversed, and (3) the channel width W (bits) which is limited by the bisection width of the wiring media divided by the channel bisection of the network[4]. The latency seen by a single message in a network with no other traffic (zero-load latency or T_0) is directly determined by these three factors:

$$T_0 = HT_n + \frac{D}{v} + \frac{L}{Wf}. \tag{1}$$

Where T_n is the propagation delay of a node (s), v is the signal propagation velocity[5] (cm/s), and f is the wire bandwidth (s^{-1}).

The three-dimensional express cube topology [12], a k-ary 3-cube with express channels added to skip intermediate hops (Figure 2B) when travelling large distances, can simultaneously optimize H, D, and λ_{sN} to achieve performance that is within a small fraction of physical and information-theoretic limits. The number of hops H is bounded by $log_q N$ if a q way decision is made at each step. The express cube network achieves this bound by inserting a hierarchy of interchanges into a k-ary n-cube network (Figure 2C). The wire distance, D is kept to within $2^{-1/3}$ of the physical minimum by always following a manhattan shortest path. Finally, the number of network channels can be adjusted to use all available wiring capacity (Figure 2D).

Figure 3 compares the performance of flat and hierarchical express cubes with a regular k-ary n-cube and a wire with no switching. The ratio of the delay of a node, $T-n$, to the delay of a wire between two adjacent nodes, $D(1)/v$, is denoted $\alpha = T_n v/D(1)$. The figure assumes $\alpha = 64$. The figure shows that a flat express cube decreases delay to a multiple of wire delay determined by the ratio α to interchange spacing, i. Interchange spacing is set to the square-root of the distance to balance the delay due to local channels with the delay due to express channels. The hierarchical cube with three levels ($l = 3$) permits small interchange spacing and allows local and global delays to be optimized simultaneously.

The advantages of minimum H and maximum λ_{sN} achieved by the express cube topology are important for very large networks. For smaller networks (less than 4K nodes), however, a simpler three-dimensional torus or mesh network, k-ary 3-cube, is usually more cost ef-

[4] In some small networks, W is constrained by component or module pinout and not by bisection width.

[5] Typically v is a fraction of the speed of light $0.3c \leq v \leq c$.

fective. The 3-D mesh also provides manhattan shortest paths in physical space to keep D near minimum, has a very regular structure, and uses uniformly short wires simplifying the electrical design of the network.

Three-dimensional networks are required to obtainin adequate throughput for machines larger than 256 nodes. As machines grow, the throughput per node varies inversely with the number of nodes in a row, as $N^{1/2}$ for a 2-D network and as $N^{2/3}$ for a 3-D network. 3-D networks provide adequate throughput up to 4K nodes (16 nodes per row). Beyond this point express cubes and/or careful management of locality is required. For machines of 256K or larger, express cubes become bisection-limited and locality must be exploited. No cost-effective network can scale throughput linearly with the size of the machine. Above a certain size, all networks become bisection-width limited and hence have a throughput that grows as $N^{2/3}$.

Routing, the assignment of a path to a message, determines the static load balance of a network. Most routers built to date have used deterministic routing – where the path depends only on the source and destination nodes. Deterministic routers can be made simple and fast, and deadlock avoidance becomes much easier. In particular, deterministic routing in dimension order permits the switch to be cleanly partitioned [17]. For some traffic patterns, deterministic routing results in a degradation in performance due to channel load imbalance. However, for most cases deterministic routing has proved adequate.

Several adaptive routing algorithms have been proposed [14, 4, 25] that are capable of dynamically detecting and correcting channel load imbalance. Adaptive routers also are able to route around a number of faulty nodes and channels. Most adaptive routers require much more complex logic than deterministic routers. The planar adaptive routing algorithm [4] is particularly attractive in that it retains much of the simplicity of dimension-order routing.

Flow control involves dynamically allocating buffer and channel resources to messages in the network. Most parallel computer networks use wormhole routing [8] in which buffers are allocated to messages while channels are allocated to flow-control digits or *flits*. To keep routers small and fast, channel buffers are often shorter than messages. Thus it is possible for a message to be blocked on the receiving side of a channel while part of the message remains on the transmitting side. With only a single buffer per channel, blocking a message on the transmitting side would idle the channel wasting network resources.

Virtual-channel flow control permits messages to pass blocked messages and make use of what would otherwise be idle channels [13]. By associating several buffers (virtual channels) with each physical channel and multiplexing them on demand, a network loaded with uniform traffic can operate at 90% of its peak channel capacity. In comparison, the throughput of a network with only a single buffer per node saturates at 20% to 50% of capacity depending on the topology and routing. Virtual channel flow control uses several small, independent buffers in place of a single large queue to more efficiently use valuable router storage. Figures 4 and 5 show the effect of adding virtual channels to the latency and throughput of 2-ary n-fly networks.

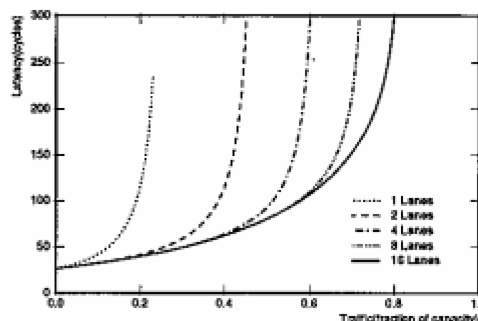

Figure 4: Latency as a function of offered traffic for a 2-ary 8-fly network with 1, 2, 4, 8, and 16 virtual channels per physical channel.

The network technology described above is able to meet the goal of providing global memory access with a latency comparable to that of a uniprocessor. Compare for example a 64-node 3-D torus with 1MByte per node with a comparably sized single processor machine with 64Mbytes. Both of these machines will fit comfortably on a desktop. Since network channels are uniformly short it is customary to operate them at twice the processor rate [10] (or more [5]). For our comparison we will use a processor rate of 50MHz and a network clock of 100MHz.

The 64-node torus requires an average of 6 hops to reach any node in the machine (HT_n=60ns). A message of six 16-bit flits (L/Wf=60ns) is sent in each direction for a read operation. The composition time of the message and the initiation of the memory access can be overlapped with this L/Wf. Thus the one-way communication time is 120ns. The memory access itself takes 100ns. Adding the reply communication time (again terminal operations are overlapped with the L/Wf time) gives a total access time of 340ns. The uniprocessor requires 1 cycle to get off chip, 2 cycles to get across a bus, and 1 cycle to initiate the memory operation (80ns total). Again the memory read itself is 100ns and the reply across the bus requires another 80ns for a total of

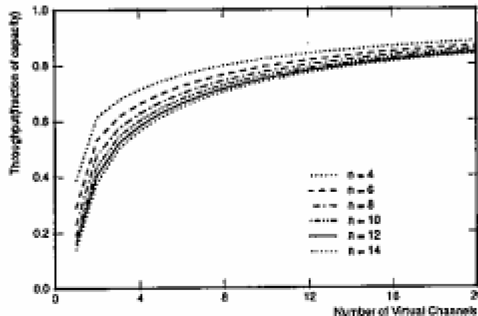

Figure 5: Throughput of 2-ary n-fly networks with virtual channels as a function of the number of virtual channels.

260ns. Thus the uniprocessor is only 80ns or 24% faster. Much of the additional delay can be attributed to the fact that the parallel computer network has more decisions to make during routing and is able to handle many messages simultaneously. While these capabilities have a slight negative affect on latency, they give a significant throughput advantage.

To see the throughput advantage, consider the problem of rotating a matrix about its center row. To perform one 64-bit move, the conventional machine requires two memory cycles or 520ns for a rate of 123Mbits/s. With an interleaved memory and a lockup-free memory interface (which few processors have) it could overlap operations to complete one every 160ns for a rate of 400Mbits/s. The parallel computer on the other hand can apply its entire bidirectional bisection bandwidth of 256 16-bit channels to the problem for a total bandwidth of 409.6GBits/s.

In summary, modern interconnection network technology gives latency comparable to conventional memory access times with throughput orders of magnitude higher. Raw network performance solves only half of the communication problem, however. To use such a network effectively requires efficient communication mechanisms.

5 Mechanisms

Mechanisms are the primitive operations provided by a computer's hardware and systems software. The abstractions that make up a programming system are built from these mechanisms [18, 9]. For example, most sequential machines provide some mechanism for a push-down stack to support the last-in-first-out (LIFO) storage allocation required by many sequential models of computation. Most machines also provide some form of memory relocation and protection to allow several processes to coexist in memory at a single time without interference. The proper set of mechanisms can provide a significant improvement in performance over a brute-force interpretation of a computational model.

Over the past 40 years, sequential von Neumann processors have evolved a set of mechanisms appropriate for supporting most sequential models of computation. It is clear, however, from efforts to build concurrent machines by wiring together many sequential processors, that these highly-evolved sequential mechanisms are not adequate to support most parallel models of computation. These mechanisms do not support synchronization of events, communication of data, or global naming of objects. As a result, these functions, inherent to any parallel model of computation, must be implemented largely in software with prohibitive overhead.

For example, most sequential machines require hundreds of instructions to create a new process or to send a message. This cost prohibits the use of fine-grain programming models where processes typically last only a few tens of instructions and messages contain only a few words. It is not hard to construct mechanisms that permit tasks to be created and messages sent in a few instruction times; however, these mechanisms are not to be found on conventional processors.

Some parallel computers have been built with mechanisms specialized for a particular model of programming, for example dataflow or parallel logic programming. However, our studies have shown that most programming models require the same basic mechanisms for communication, synchronization, and naming. More complex model-specific mechanisms can be built from the basic mechanisms with little loss in efficiency. Specializing a machine for a particular programming model limits its flexibility and range of application without any significant gain in performance. In the remainder of this section, we will examine mechanisms for communication, synchronization, and naming in turn.

Communication between two processing nodes involves the following steps:

1. *Formatting:* gathers the message contents together.
2. *Addressing:* selects the physical destination for the message.
3. *Delivery:* transports the message to the destination.
4. *Allocation:* assigns space to hold the arriving message.
5. *Buffering:* stores the message into the allocated space.

6. *Action:* carries out a sequence of operations to handle the message.

All programming models use a subset of these basic steps. A shared memory read operation, for example, uses all six steps. A read message is formatted, the address is translated, the message is delivered by the network, the message is buffered until the receiving node can process it, and finally a read is performed and reply message is sent as the action. Some models, such as synchronous message passing always send messages to preallocated storage and thus omit allocation (step 4). In some cases, no action is required to respond to a message and step 6 can be omitted.

The `SEND` instruction, first used in the message-driven processor [15, 16], with translation of destination addresses [19] efficiently handles the first two steps: formatting and addressing. A message is sent with a sequence of `SEND` instructions followed by a `SENDE` instruction. A `SEND` instruction takes a number of arguments equal to the number of read register ports (typically two) and appends its arguments to a message. A `SENDE` instruction is identical to the `SEND` except that it also signals the end of the message. The first `SEND` after a `SENDE` starts a new message. By making full use of the register bandwidth the `SEND` instruction reduces formatting overhead to a minimum. The alternative approaches of formatting a message (1) in memory or (2) by writing to a memory mapped network port have much lower bandwidth and higher latency.

Translation is achieved by interpreting the first word of the message stream (the first argument of the first `SEND`) as a virtual destination address and translating it to a physical address when a message is sent. A simple translation-lookaside buffer (TLB) efficiently performs this translation. This approach of translating virtual network addresses to physical addresses during the `SEND` operation permits message sends from user code to be fully protected without incurring the overhead of a sytem call (as is done on many machines today). User code is only permitted to send messages to addresses that are entered in TLB. Sending a message to any other address raises an exception.

Communication operations that do not require allocation and or remote action can use a subset of the basic mechanism. A remote write operation, for example, requires neither of these functions. Avoiding allocation and action in this case eliminates the overhead of copying the message from newly allocated storage to its final destination. The first `SEND` instruction of a message can specify whether allocation (`.A` suffix) and/or spawning a task (`.S` suffix) are required [19]. A `SEND` with no suffix would simply perform a remote write, `SEND.A` would allocate but not initiate a remote action, and `SEND.SA` would do both. The sending node treats these three `SEND` operations identically and simply sends along the two option bits with the message. The receiving node examines the option bits to determine whether allocation and/or action is required. If an action is required, the routine to be invoked is specified by the second word of the message.

Storage allocation and message buffering must be performed in hardware to achieve adequate performance. While approaches using stack (LIFO) or queue (FIFO) based storage are simple to implement [10], they may require copying if messages are not deallocated in order. An alternative is to allocate message buffers off a free list of fixed-sized segments [40]. Management of such a free list is simple (only a single pointer is required) and it does not restrict message lifetimes. Messages too long for the fixed-sized segments can be handled in an overflow area.

With any allocation scheme, a method for handling message buffer overflow is required. Because handling an overflow may require access to other nodes, the network must be usable even when a full buffer is causing messages to back up into the network. This is accomplished on the J-Machine by using two virtual networks [10]. The actual overflow handling may be performed in software as it is a rare event. While many strategies may be used to handle overflow, a simple one is to return overflowing messages to their senders. With this scheme each node must guarantee that it has storage to hold each message it originates until it is acknowledged.

The final step of a communication operation is to initiate a remote action by creating and dispatching a task. A task or process consists of a thread of control and an addressing environment. A thread can be created in a few clock cycles by loading a processor's IP to set the thread of control and initializing its memory management registers to alter the addressing environment. On the J-Machine, each message in the message queue is treated as a thread that is ready to run and threads are dispatched when they reach the head of the queue This dispatching on message arrival also serves as the basis of a synchronization mechanism.

Synchronization enforces an ordering of events in a program. It is used, for example, to ensure that one process writes a memory location before another reads it, to provide mutual exclusion during critical sections of code, and to require all processes to arrive at a barrier before any processes leave.

Any synchronization mechanism requires a namespace that processes use to refer to events, a method for signalling that an event is enabled, and a method for forcing a processor to wait on an event. Using tags for synchronization, as with the presence bits on the HEP [36], uses the memory address space as the synchronization namespace. This provides a large synchronization

namespace with very little cost as the memory manangement hardware is reused for this function. It also has the benefit that when signaling the availability of data, the data can be written and the event signaled in a single memory operation. Since it naturally signals the presence of data, we refer to this synchronization using tags on memory words as *data synchronization*[40].

With synchronization tags, an event is signaled by setting the tag to a particular state. A process can wait on an event by performing a synchronizing access of the location which raises an exception if the tag is not in the expected state. A synchronizing access may optionally leave the tag in a different state. Simple producer/consumer synchronization can be performed using a single state bit. In this case, the producer executes a synchronizing write which expects the tag to be empty and leaves it full. A synchronizing read which expects the location to be full and leaves it empty is performed by the consumer. If the operations proceed in order, no exceptions are raised. An attempt to read before a write or to write twice before a single read raises a synchronization exception. More involved synchronization protocols require additional states (for example to signal that a process is waiting on a location) [19].

The communication mechanism described above complements data synchronization by providing a means for a process on one node to signal an event on a remote node. In the simplest case, a message handler can perform a synchronizing read or write operation. However, it is often more efficient to move some computation to the node on which the data is resident. Consider for example the problem of adding a value to a remote location[6]. One could perform a remote synchronizing read that marks the location empty to gain exclusive access, perform the add, and then perform a remote synchronizing write. Sending a single message to invoke a handler that performs the read, add, and write on the remote node, however, reduces the time to perform the operation, the number of messages required, and the amount of time the location is locked.

Many machines have implemented some form of global barrier synchronization. For example, the Caltech Cosmic Cube [32] had four program accessible wire-or lines for this purpose. While global barrier synchronization is useful for some models, it can be emulated rapidly using communication and data synchronization. If there is sufficient slack time from when a process signals that it has reached the barrier to when it waits on the barrier, this emulation will not affect program performance. The required amount of slack time varies logarithmically with the number of processors performing the barrier. Also, the major use of barrier synchronization (inserting a barrier between code that produces a structure (e.g., array) and code that consumes the structure) is eliminated by data synchronization. By synchronizing in the data space on each individual element of the data structure, control space synchronization on the program counter between the producer and consumer is neither required nor desired. It is more efficient to allow the producer and consumer to overlap their execution subject to data dependency constraints. Barrier synchronization mechanisms also have the disadvantage that they require a separate namespace which tends to be small because of the prohibitive cost of providing many simultaneous barriers, and they consume pin and wire resources that could otherwise be used to speed up the general communication network.

[6]This occurs for example when performing LU decomposition of a matrix.

The mechanism that enforces event ordering solves only half of the synchronization problem. Efficient synchronization also requires an *agile* processor that can rapidly switch processes and handle events and messages to reduce the exception handling and context switching overhead when switching processes while waiting on an event. Rapid task switching can be provided by providing multiple register sets or a named-state register set [29]. Exception handling is accelerated by specifically vectoring exceptions, providing separate registers for exception handling, and explicitly passing arguments to exception handlers [19].

6 Experience

In the Concurrent VLSI Architecture Group at MIT, we have built the J-Machine [10], a prototype fine-grain parallel computer with a high-speed network and efficient yet general communication and synchronization mechanisms. The J-Machine was built to test and evaluate our ideas on mechanisms and networks, as a proof of concept for this class of machine, and as a testbed for parallel software research. Small prototoypes have been operational since June of 1991. We expect to have a 1024-processor J-Machine on-line during the summer of 1992.

The J-Machine communication mechanism permits a node to send a message to any other node in the machine in $< 1.5\mu s$. On message arrival, a task is created and dispatched in 200ns. A translation mechanism supports a global virtual address space. These mechanisms efficiently support most proposed models of concurrent computation and allow parallelism to be exploited at a grain size of 10 operations. The hardware is an ensemble of up to 65,536 nodes each containing a 36-bit processor, 4K 36-bit words of on chip memory, 256K words of DRAM, and a router. The nodes are connected by a high-speed 3-D mesh network with deterministic dimension order routing. The J-Machine has about the grain

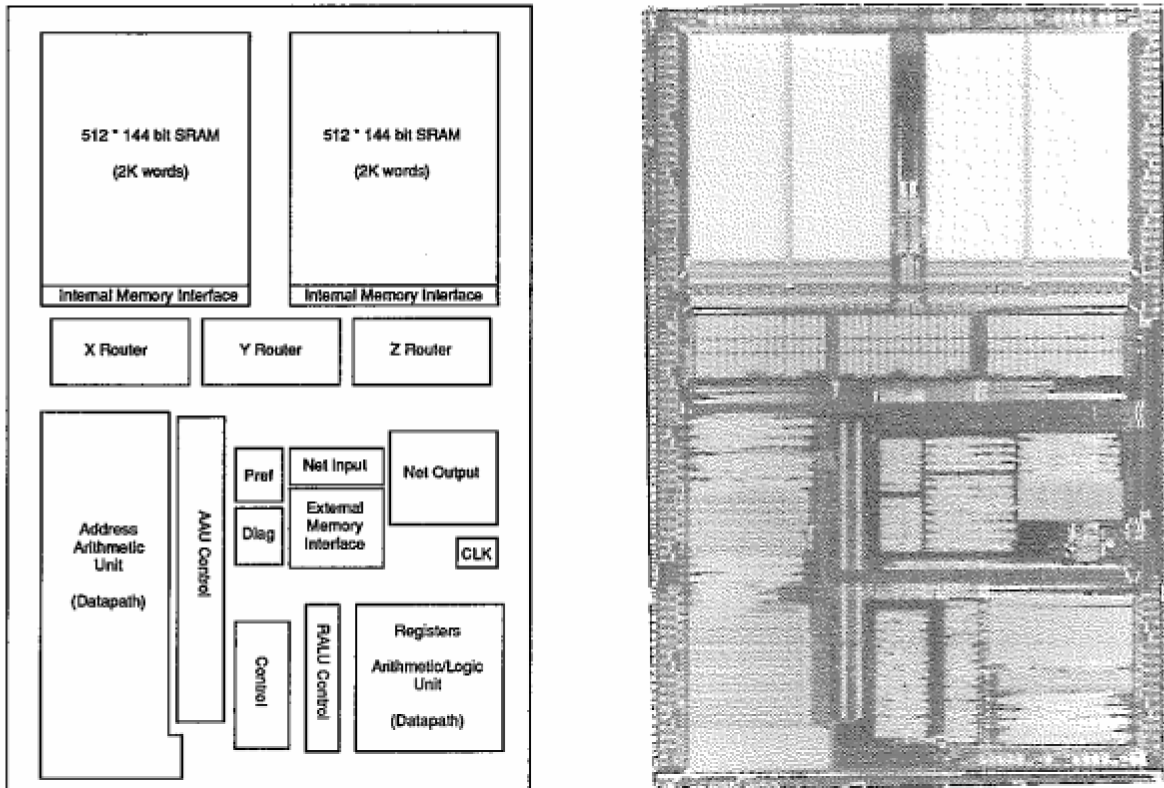

Figure 6: Floorplan and Photograph of a Message-Driven Processor chip.

size of the cost-balanced machine described in Section 3, one processor per megabyte of memory.

A photograph of the message-driven processor chip used in the J-Machine is shown in Figure 6. One of these chips combined with three external DRAM parts forms a J-Machine node. An array of 64 nodes is packaged on a single board (Figure 7). These boards are stacked and connected side-to-side to form larger J-Machines.

Three software systems are currently operational on the J-Machine. It runs Concurrent Smalltalk (CST) [24], a version of Id based on the Berkeley TAM system [37, 6], and a dialect of "C". Execution of these diverse programming systems has demonstrated the efficiency and flexibilty of the J-Machine mechanisms.

Table 1 shows the advantage of efficient mechanisms. The left column of the table lists the operations involved in performing a remote memory reference on a 1024-node parallel computer. The next two columns list the approximate number of instruction times required to perform each operation on the Intel Paragon [5] and

Operation	Paragon	J-Machine	Ideal
Send 4-Word Message	600	3	2
Network Delay	32	10	10
Buffer Allocation	20	0	0
Switch To Handle Msg	1000	10	1
Presence Test	5	0	0
Send 3-Word Return Msg	600	3	2
Network Delay	32	10	10
Buffer Allocation	20	0	0
Switch To Handle Msg	1000	3	1
Switch To Restart Task	1000	10	1
TOTAL	4309	49	27

Table 1: The time to perform a remote memory reference on the Intel Paragon, a conventional message-passing multicomputer, the J-Machine, a fine-grain parallel computer, and the time that could be achieved with current technology (Ideal). Switch refers to a task switch.

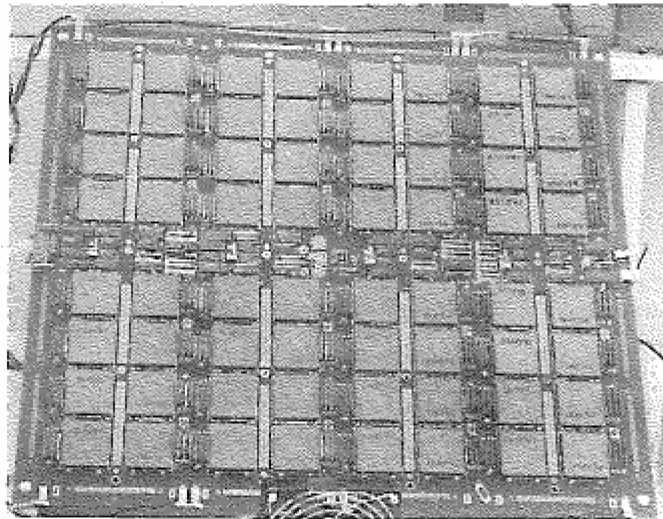

Figure 7: Photograph of a 64-node J-Machine board.

the J-Machine. Many of these times were derived from the study reported in [38]. The final column of the table shows the times that could be achieved with techniques that are currently understood.

The table shows that while both machines have fast networks the time to carry out a simple remote action is many times greater on the conventional machine. The single largest contributor is the task switching time[7]. The overhead of task switching in a conventional operating system is unacceptable in this environment. Even if the task switch time were reduced to zero, the overhead of sending a message[8] in a system where this function is handled in software is still prohibitive. End-to-end hardware support for communication is required to achieve acceptable latency.

The rightmost column represents times that could be achieved by making some minor modifications to the J-Machine. In particular, task switch time could be reduced from 10 cycles (when registers need to be saved) or 3 cycles (w/o register save) to a single cycle by providing more support for multithreading [29, 39]. The J-Machine would also benefit from more user registers, automatic destination translation on message send, being able to subset the communication operation, and a non-LIFO message buffer.

[7]The estimate of 1000 instruction times or 25μs for the i860 is extrapolated from other microprocessors and hence very generous; because of the complexity of event handling on this chip, the actual number is higher.

[8]Some receive time is also included in this number.

7 Related Work

Like the message-driven processor from which the MIT J-Machine is built, the Caltech MOSAIC [33], Intel iWARP [3], and INMOS Transputer [26] are integrated processing nodes that incorporate a processor with memory and communication on a single chip. These integrated nodes, however, lack the efficient mechanisms of the MDP and thus cannot efficiently support many different models of computation. Also, the software-routed, bit-serial Transputer network does not have adequate performance for many applications.

Many machines built for a specific model of computation have been generalizing their mechanisms. For example, the MIT Alewife machine [1], while specialized for the shared-memory model, provides an interprocessor interrupt facility that can be used for general message-passing. Being memory mapped, this operation is somewhat slower than the register-based send operation described above. Dataflow machines, which once hard-wired a particular dataflow model into the architecture [30, 34], have also been moving in the direction of general mechanisms with the EM4 [31] and *T [28].

8 Conclusion

Two enabling technologies, fast networks (Section 4) and efficient interaction mechanisms (Section 5), make it possible to build and program fine-grain parallel computers. Fine-grain machines have much less memory per processor than conventional machines because they are balanced by cost, rather than by capacity to speed ratios. Increasing the processor to memory ratio improves the processor throughput and local memory bandwidth by a factor of 50 with only a small increase in system cost.

We expect this dramatic performance/cost advantage will lead to mechanism-based fine-grain parallel computers becoming universal, replacing sequential computers in all sizes of systems from personal desktop computers to institutional supercomputers. This universal parallel computer will not happen with existing semiconductor price structures, where processor silicon is an order of magnitude more expensive per unit area than memory silicon. Cost effective fine-grain computing requires a true *jellybean* (inexpensive and plentiful) processing-node chip.

Low-latency networks enable each node in a fine grain machine to access any memory location in the machine in time competitive with a global memory access in a conventional machine. Thus, the small memory per node does not limit either the problem size that can be handled or sequential execution speed. A fine-grain machine can execute sequential programs with performance competitive with conventional machines.

High-bandwidth networks and efficient interaction mechanisms enable fine-grain computers to apply their high aggregate processor throughput and memory bandwidth with minimum overhead. Reducing interaction overhead to a few instruction times (Table 1) increases the amount of parallelism that can be economically exploited. It also simplifies programming as tasks and data structures no longer have to be grouped into large chunks to amortize large communication, synchronization, and task-switching overheads.

At MIT we have built and programmed the J-Machine to test, evaluate, and demonstrate our network and mechanisms. By running three programming systems on the machine, we have demonstrated the flexibility of its mechanisms and generated some ideas on how to improve them. The next step is to work to commercialize this technology by developing a more integrated and higher-performance processing node in today's technology and by providing bridges of compatibiliity to existing sequential software.

References

[1] Anant Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[2] David Bailey. The NAS Parallel Benchmarks. Presentation given in 1991.

[3] Shekhar Borkar et al. iWARP: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–338. IEEE, November 1988.

[4] Andrew A. Chien and Jae H. Kim. Planar-Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, Queensland, Australia, May 1992. IEEE.

[5] Intel Corporation. Paragon XP/S. Product Overview, 1991.

[6] David E. Culler et al. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175. ACM, April 1991.

[7] William J. Dally. Directions in Concurrent Computing. In *Proceedings of the International Conference on Computer Design*, pages 102–106. IEEE, October 1986. Conference at Port Chester, New York.

[8] William J. Dally. Wire-Efficient VLSI Multiprocessor Communication Networks. In Paul Losleban, editor, *Proceedings of Stanford Conference on Advanced Research in VLSI*, pages 391–415. MIT Press, 1987.

[9] William J. Dally. Mechanisms for Concurrent Computing. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 154–156, December 1988.

[10] William J. Dally. The J-Machine System. In Patrick Winston with Sarah A. Shellard, editor, *Artificial Intelligence at MIT: Expanding Frontiers*, chapter 21, pages 536–569. MIT Press, 1990.

[11] William J. Dally. Network and Processor Architecture for Message-Driven Computers. In Suaya and Birtwhistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann, 1990.

[12] William J. Dally. Express Cubes: Improving the Performance of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, pages 1016–1023, September 1991.

[13] William J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), March 1991.

[14] William J. Dally and Hiromichi Aoki. Adaptive Routing using Virtual Channels. *IEEE Transactions on Parallel and Distributed Computing*, 1992.

[15] William J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–205. IEEE, June 1987.

[16] William J. Dally et al. Design and Implementation of the Message-Driven Processor. In *Proceedings of the 1992 Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*. MIT Press, March 1992.

[17] William J. Dally and Paul Song. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *Proceedings of the International Conference on Computer Design*, pages 230–234. IEEE, October 1987.

[18] William J. Dally and D. Scott Wills. Universal Mechanisms for Concurrency. In G. Goos and J. Hartmanis, editors, *Proceedings of PARLE-89*, pages 19–33. Springer-Verlag, June 1989.

[19] William J. Dally, D. Scott Wills, and Richard Lethin. Mechanisms for Parallel Computing. In *Proceedings of the NATO Advanced Study Institute on Parallel Computing on Distributed Memory Mutliprocessors*. Springer, 1991.

[20] J. A. Fisher and B. R. Rau. Instruction-Level Parallel Processing. *Science*, pages 1233–1241, September 1991.

[21] Geoffrey Fox et al. *Solving Problems on Concurrent Computers*. Prentice Hall, 1988.

[22] John L. Hennessy and Patterson David A. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, San Mateo, 1990.

[23] John L. Hennessy and Norman P. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *Computer*, pages 18–29, September 1991.

[24] Waldemar Horwat, Andrew Chien, and William J. Dally. Experience with CST:Programming and Implementation. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, 1989.

[25] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. In *18th Annual Symposium on Computer Architecture*, pages 212–221, 1991.

[26] InMOS Limited. IMS T424 Reference Manual. Order Number 72 TRN 006 00, November 1984.

[27] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass, 1980.

[28] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. Computation Structures Group Memo 325-1, Massachusetts Institute of Technology Laboratory for Computer Science, November 15 1991.

[29] Peter R. Nuth and William J. Dally. A Mechanism for Efficient Context Switching. In *Proceedings of the International Conference on Computer Design*. IEEE, October 1991.

[30] Gregory M. Papadopoulos and David E. Culler. Monsoon: an Explicit Token-Store Architecture. In *The 17th Annual International Symposium on Computer Architecture*, pages 82–91. IEEE, 1990.

[31] S. Sakai et al. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 46–53, 1989.

[32] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[33] Charles L. Seitz et al. Submicron Systems Architecture. Semiannual Technical Report Caltech-CS-TR-90-05, Department of Computer Science, California Institute of Technology, March 15 1990.

[34] Toshio Shimada, Kei Hiraki, Kenji Nishida, and Satosi Sekiguchi. Evaluation of a Prototype Data Flow Processor of the Sigma-1 for Scientific Computations. In *13th Annual International Symposium on Computer Architecture*, pages 226–234. IEEE, June 1986.

[35] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[36] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Vol. 298 Real-Time Signal Processing IV*, pages 241–248. Denelcor, Inc., Aurora, Col, 1981.

[37] Ellen Spertus and William J. Dally. Experiments with Dataflow on a General-Purpose Parallel Computer. In *Proceedings of International Conference on Parallel Processing*, pages II231–II235, Aug 1991.

[38] Brian Totty. Experimental Analysis of Data Management for Distributed Data Structures. Master's thesis, University of Illinois, 1991.

[39] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *The 16th Annual International Symposium on Computer Architecture*, pages 273–280. IEEE Computer Society Press, 1989.

[40] D. Scott Wills. *Pi: A Parallel Architecture Interface for Multi-Model Execution*. PhD thesis, Massachusetts Institute of Technology, May 1990.