THE DESIGN OF PROCESSING ELEMENTS ON A MULTIPROCESSOR SYSTEM WITH A HIGH-BANDWIDTH, HIGH-LATENCY INTERCONNECTION NETWORK

Richard Kenner, Susan Dickey, and Patricia J. Teller
kenner@nyu.edu, dickey@nyu.edu, teller@nyu.edu

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012
(212) 998-3339
Ultracomputer Note #157
January, 1989

Abstract

The environment of a highly-parallel, shared-memory multiprocessor system with a high-bandwidth, high-latency interconnection network is different in many ways from that encountered in a uniprocessor system. We describe these differences and the impact they have on the design of processing elements. In addition, this paper describes methods that can be used to evaluate the impact of architectural choices on the performance of any system that uses a similar network. Two detailed designs of processing elements, one using a CISC processor and the other using a RISC, are given as examples.

1. Introduction

A highly-parallel, shared-memory multiprocessor (HPSMM) is composed of thousands of processing elements (PEs) connected via a multistage interconnection network to an equal number of memory modules (MMs). This paper discusses the effects of the processor-memory interconnection network on the design of the PEs, in the context of a particular HPSMM architecture, the NYU Ultracomputer. Both the NYU Ultracomputer architecture and its processor-memory interconnection network have been described in many forums (e.g., see [10] and [14]).

The NYU Ultracomputer project has built a number of prototypes. The latest bus-based system, called Ultra II, contains eight Motorola MC68010s sharing 8MB of memory and is used for application and operating system development. We are currently completing the design of the next prototype, Ultra III, which will contain 64 PEs. Unlike earlier prototypes, Ultra III will have a PE-MM interconnection network built from custom VLSI chips.

The behavior of the network affects the design of the PEs and MMs. Specifically, the network has a very high bandwidth but imposes a latency of several microseconds on each memory request. This environment is significantly different from that of conventional uniprocessors and produces a different set of tradeoffs. For one thing, it is not as important to reduce the PE's request rate as it is on a uniprocessor since it only has an indirect effect on latency. Instead, a PE should try to issue as many requests as it can as early as possible to compensate for the relatively high latency of the network. The methods presented in this paper to deal with the high latency and exploit the available high bandwidth are applicable to many HPSMM systems.

In order to study the impact of architectural choices on system performance, we use both simulation and analysis to determine the behavior of the network and memory subsystems. In addition, a program that includes an analytical model of the network models the complete architecture as a closed-loop system. The PE design issues discussed below were motivated by the understanding of HPSMM behavior gained through these studies.

In the following sections, we use the term *processor* to refer to the actual computational element (usually a microprocessor) that is part of the PE. A *PE* consists of a processor, an interface to the interconnection network, caches, and possibly other auxiliary devices. Similarly, an *MM* includes memory chips, control logic, and an interface to the network.

This paper begins with a review of the NYU Ultracomputer architecture and the processor-memory interconnection network. Our system modeling procedures and results are then

This work was supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under contract DE-AC02-76ER03077, in part by the National Science Foundation under grant DCR-8413359, and in part by I.B.M. under joint study agreement N00039-84-R-0605(Q).

described. Issues involved in PE design and possible general design approaches are next discussed. Finally, two examples of PE design are presented, one using the Motorola MC68030 (for which architectural simulation was run) and one using the AMD Am29000 CPU (which has been chosen for Ultra III).

2. The NYU Ultracomputer Architecture

The Ultracomputer architecture, shown in Figure 1, is designed to scale to thousands of PEs. Unlike an idealized multiprocessor, called a *paracomputer* by [25] and classified as a WRAM by [1], it does not assume single-cycle access to shared memory but instead connects the PEs to an equal number of MMs via a multistage interconnection network built from custom VLSI switches that merge simultaneous requests for the same memory location.

The processor's instruction set is augmented by fetch-and-ϕ operations (which can be defined for any associative function ϕ), each of which atomically modifies a memory location and returns its old value. For example, fetch-and-add (F&A) is an indivisible add to memory: F&A(X, e), where X is an integer variable and e is an integer expression, returns the (old) value of X and replaces X in memory with the sum $X + e$. Using fetch-and-ϕ, it is possible to write efficient parallel algorithms with reduced synchronization overhead. Early versions of such algorithms are given in [15]; see [30] for a large collection of such algorithms.

3. The Interconnection Network

The Ultracomputer network (shown in Figure 2 for 8 PEs) has the topology of Lawrie's Ω network [18] and the SW Banyan of [13]. It consists of log N stages, where N is the number of PEs in the system. In this paper, we discuss a network with $\log_2 N$ stages, each having $N/2$ nodes, where a node is a 2×2 switch that routes requests from PEs to MMs and responses (to these requests) from MMs to PEs. A switch has four bidirectional ports, two connecting either to PEs or switches closer to the PEs and two connecting either to MMs or switches closer to the MMs. The switch is buffered in order to avoid the bandwidth limitation of unbuffered networks [16].

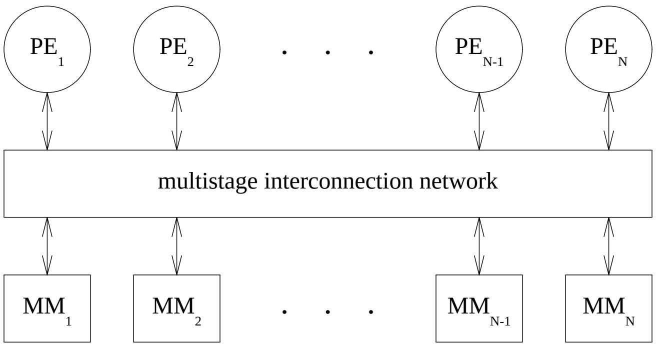

Figure 1. Ultracomputer Architecture

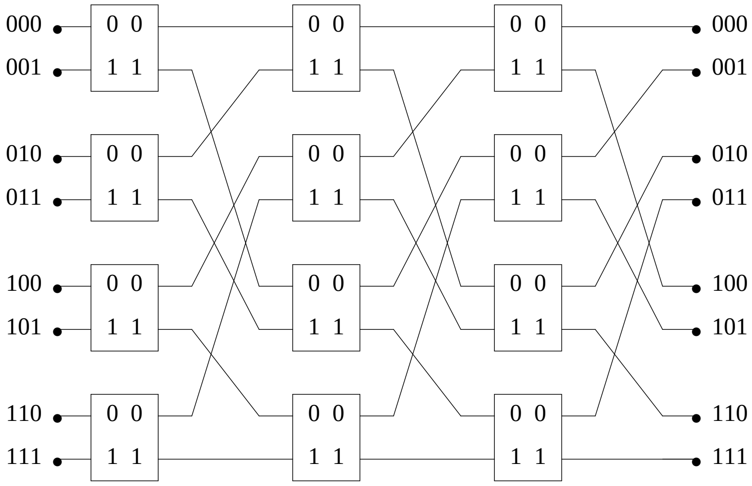

Figure 2. Ultracomputer Network (N = 8)

3.1 Switch Structure

A switch consists of two independent routers, one for requests and the other for responses; each contains four queues and two multiplexers, as shown in Figure 3a (see [5], [6], and [7] for a more detailed presentation). Most network analysis and modeling assumes a slightly different structure, shown in Figure 3b, where each router consists of two 2-input, 1-output queues. We treat these structures as equivalent in our summary of switch behavior; see [8] for a more precise presentation.

The switch can *combine*, or merge, two requests that refer to the same memory location and that are simultaneously present in its queues. Combining reduces network saturation resulting from *hot-spot* traffic [24], characterized by many concurrent requests to the same location. Since this is a common access pattern for parallel programs performing synchronization operations, combining is important for adequate system performance.

These switches can potentially combine any fetch-and-ϕ operation[1], although we only plan

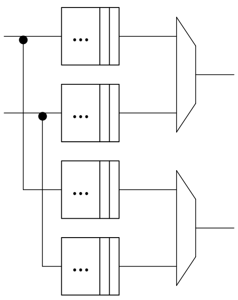

Figure 3a.

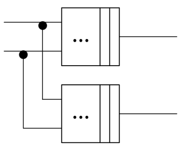

Figure 3b.

Two versions of a 2×2 switch. Figure 3a is used in the Ultracomputer, while Figure 3b is an alternate structure often used in network analysis.

[1] *Loads* and *stores* are also fetch-and-ϕ operations. In a load, ϕ is the projection operation π_1, and in a store, ϕ is

to implement combining for some of them. When two messages destined for the same location meet at a switch, one is saved in a *wait buffer* and a merged message is sent towards the MMs. When a switch recognizes that an MM response matches a message in its wait buffer, a second response is generated in order to satisfy both requests. Other than the wait buffer, there is no connection between the forward path, from PEs to MMs, and the return path, from MMs to PEs.

A 32-bit store requires a minimum message size of 64 bits (32 for address and 32 for data). Options range from bit-serial transmission to 64-bit ports. Larger packet sizes produce higher network bandwidth but increase the difficulty of packaging the network. The systolic design of the queues requires that messages be composed of an even number of packets [7]; the combining logic is simplified by placing if the address and operation code, needed to decide if two requests can be combined, in the first packet of the message. The second requirement imposes a minimum width; the first makes a larger width of limited utility. Therefore, 32-bit requests are transmitted as two packets; the first containing the address, the second the data.

Messages are pipelined through the network: switches transmit each packet of a message as soon as possible rather than waiting for the entire message. All packets of a given message arrive at a MM consecutively and in order. The minimum one-way network transit time, in the absence of any delay due to routing conflicts, is thus equal to the sum, rather than the product, of the number of stages and the number of packets per message.

3.2 Message Format

In order to match a response from memory with a message in a wait buffer, each message in the network must be uniquely identifiable by its first (address) packet. The PE number (actually the bits of the PE number representing the path to the switch) is one part of such a key. To allow each PE to have more than one request outstanding in the network, other information must be included. The memory address is already present in the message and can be used for this purpose. However, this would mean that the PE must be prevented from issuing multiple outstanding requests to the same location. Although the average reduction in the number of outstanding requests is not significant, the amount of extra logic to perform the required bookkeeping and comparisons is.

An alternate method, due to the IBM RP3 project [23], augments each request from a PE with a unique value, called an *outstanding request index* (ORI); thus the processor must only track the indices in use by outstanding requests. Two further advantages of this method are that the address is not transmitted on the return path and the wait buffer has fewer bits to compare in its associative search. The disadvantage is that additional bits must be transmitted in the forward path.

4. System Performance Modeling

Highly-parallel, shared-memory multiprocessors with interconnection networks have been modeled extensively (see [8], [16], [17], and [22]). Since we know of no exact analysis that accurately models the effects of finite buffer size and non-uniform traffic patterns, we have carried out simulations of round-trip network traffic, from the PEs to the MMs and back, to get better information about network latency and throughput. However, for estimates of cache and memory access delays and their effect on processor utilization in a specific PE design, we rely on a network latency calculation based on uniform traffic patterns and unbounded buffering at each queue, as developed of [17]. We include these calculations in a modeling program similar to that

---

π_2.

of [22] that assumes that the system reaches a steady state; the processor utilization calculation follows that of [19].

4.1 Network Simulations

We simulate networks composed of 2×2 switches under various assumptions of switch architecture, PE request generation, and memory behavior. Buffer size and structure at switches can be varied. Different clear-to-send protocols can be tested, and different combining alternatives corresponding to different hardware implementations can be specified. Output statistics include average latency, average bandwidth, average queue length per stage and number of combines that occur.

In the results shown here, PE requests were generated with some fixed probability (the *offered load*) whenever queues at the first stage of the network were not full and a message was not already in the process of being generated. The *effective throughput* is defined as the average number of messages actually generated per cycle and is less than the offered load. For two-packet messages, effective throughput cannot be greater than 50 percent. PE request generation can also simulate a processor which is allowed only a small number of requests outstanding before it must stop generating requests. In this case, the effective throughput is even smaller.

The memory is modeled as having distinct cycle and access times. The cycle time determines whether or not the memory accepts a message; the access time is used in computing total latency from PE to MM. Cycle and access times may vary for loads, stores, and other operations. The routing reversal at the memory makes delays on the forward and return paths asymmetric, and must be included for accurate system modeling.

4.2 Network Latency

Network latency has three components: the basic transit time through each network stage, the pipeline-fill time for the packets in a message after the first, and the waiting time in the queues at each stage. From [17], assuming unbounded buffer size and uniform distribution of memory addresses, the expected value of the waiting time in the first stage is

$$t\left[\frac{m^2 p(1-1/km)}{2(1-mp)}\right]$$

and the waiting time at later stages is approximately

$$t(1 + \frac{4mp}{5k})\left[\frac{m^2 p(1-1/k)}{2(1-mp)}\right],$$

where t is the cycle time of the switch, m is the number of packets per message, k is the size of the crossbar, p is the offered load in messages per cycle, and the correction factor in the second formula is obtained from simulations (see also [22] for a similar correction factor). Note the quadratic term in m. It is known that even moderately large values of m have a large effect on the *variance* of the latency [16].

The performance penalty resulting from using messages with many packets does not seem to be widely appreciated in the literature (e.g., see [12]). If data path width is reduced and crossbar size is enlarged by increasing the number of packets per message, the number of stages decreases. However, the pipeline-fill time increases and the waiting time per stage grows by a larger factor than that by which the number of stages is reduced.

We are interested in how close the above estimates are when buffer size is bounded and traffic patterns are not uniform. Due to the nature of our queues' systolic design, they do not have a fixed size, since they can hold between 6 and 14 packets when they are declared ''full''.

Simulations have shown that a queue of that size has an effective throughput very close to the offered load, up to an offered load of about 0.3 messages per cycle for two-packet messages (see Figure 4a). Round-trip latency is less than that predicted; this seems to be due to smaller queueing delays on the return path (see Figure 4b).

Our simulations show that combining keeps latency low even with heavy hot-spot traffic in a 64-PE network (see Figure 5). However, we do not know the effect of clustered traffic to memory modules or ‘‘bursty’’ traffic.

Offered Load	Infinite Queues (no blocking)	Finite queues (14 packets)	
	Predicted Latency	Throughput	Latency
0.1	17.6	0.09	16.9
0.2	25.9	0.17	18.3
0.3	45.7	0.23	20.4
0.4	111.2	0.29	23.4
0.5	1275.4*	0.33	26.9
0.6	∞	0.35	32.2
0.7	∞	0.36	35.3
0.8	∞	0.36	36.7
0.9	∞	0.36	37.4
1.0	∞	0.36	37.8

Figure 4a. Simulation of finite queues vs. analysis of infinite queues: 64 PEs, 2×2 switches, and two-packet messages. Offered load and throughput are in messages per network cycle; latency in network cycles.

*This latency for offered load of 0.49; at 0.5, the queuing system is at capacity and the predicted latency becomes infinite.

Offered Load	Stage: PE to MM					
	1	2	3	4	5	6
0.1	1.10	1.11	1.12	1.13	1.12	1.12
0.3	1.39	1.46	1.49	1.49	1.51	1.50
0.5	1.92	2.14	2.21	2.25	2.24	2.21
0.7	3.19	3.76	3.93	3.98	3.98	3.91
Offered Load	Stage: MM to PE					
	1	2	3	4	5	6
0.1	1.08	1.08	1.08	1.08	1.07	1.06
0.3	1.28	1.33	1.36	1.37	1.35	1.29
0.5	1.63	1.80	1.88	1.91	1.87	1.73
0.7	2.31	2.82	2.99	3.07	2.99	2.67

Figure 4b. Stage by stage delay in network cycles: 64 PEs, 2×2 switches, 20 packet queues, two-packet messages, and memory cycle time of two network cycles.

Hotspot Rate	Throughput (messages per cycle)				
	20%	40%	60%	80%	100%
0%	0.167	0.285	0.370	0.385	0.388
2%	0.167	0.220	0.220	0.220	0.220
5%	0.122	0.122	0.122	0.122	0.122
Hotspot Rate	Latency (network cycles)				
	20%	40%	60%	80%	100%
0%	18.26	23.27	32.45	41.21	43.05
2%	18.82	36.69	41.28	42.86	43.59
5%	48.74	56.43	58.23	59.27	59.79

Figure 5. Hotspot vs. no-hotspot traffic for a 64-PE combining network (simulation) as a function of offered load.

4.3 Multiple Outstanding Requests

The above analyses model a PE as an infinite source of requests. This is not completely accurate: in many cases, a PE may not issue another request until a previous request has been satisfied. We have run simulations that require a PE to wait for a previous load request to complete before making another, that is, a PE is not allowed to issue load request *i* until load request *i-1* completes. If one prefetch is allowed, a PE can issue load request *i* when *i-2* completes, and similarly for additional prefetching. Figure 6 shows effective throughput and latency vs. offered load for various amounts of load prefetch, with the PE not required to wait for stores. In this context, the ratio of effective throughput to offered load corresponds roughly to processor utilization. Note that as the load increases, more prefetching greatly improves performance.

4.4 Modeling Processor Utilization

To aid in making decisions about memory and cache design, a program was written to study the effect of design choices on overall processor utilization in the presence of network latency. Because it is easier to predict effects of high memory latency in the MC68030, it was used as the microprocessor in the system modeling. (Because of its prefetch capability, the Am29000 is better able to tolerate long memory delays and should have higher utilization than the 68030.) Figure 7 shows the model of the request flow between the system components used by this program.

Offered Load	Effective throughput by amount of prefetch				Latency by amount of prefetch			
	None	One	Two	∞	None	One	Two	∞
0.1	0.05	0.08	0.09	0.09	15.8	16.2	16.4	16.5
0.3	0.07	0.13	0.17	0.23	16.2	17.2	18.3	20.4
0.5	0.08	0.15	0.20	0.33	16.3	17.7	19.2	27.5
0.7	0.08	0.15	0.21	0.41	16.4	17.9	19.7	43.4

Figure 6. Effective throughput and latency for varying prefetch and offered load: 64 PEs, 2×2 switches, 20 packet queues, two-packet messages, and memory cycle time of two network cycles. Request mix: 60% loads, 40% stores. No prefetch limit on stores.

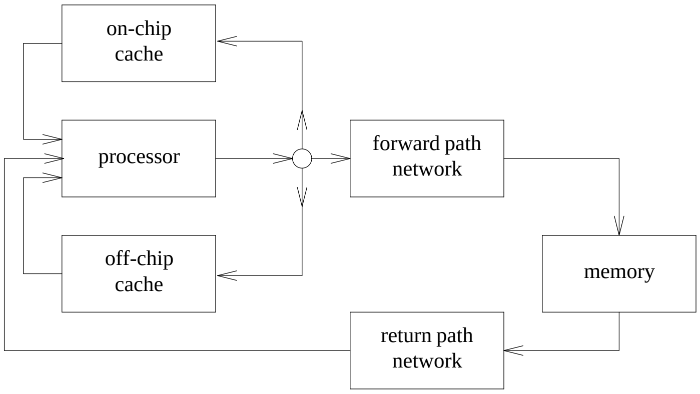

Figure 7. Program Model

Waiting for responses to memory requests transmitted through the network causes the PE to lower its request rate, which is the offered load to the network. Lowering the load decreases the latency and allows the load to increase again. Assuming that the offered load and latency reach a steady state, a ‘‘fixed point’’ (see [22]) can be reached where the expected latency yields a processor utilization consistent with the request rate.

The various components are modeled separately, which makes the total delay somewhat optimistic. Loading on the network is determined from cache hit rates, which are specified separately for internal and external, I- and D-caches. No delay is assumed for on-chip cache hits; off-chip cache hits are modeled with a varying number of wait states. Forward and return network paths are modeled using the formulas from [17] with a different average message size for each direction, depending on the assumed request mix. The processor does not wait on stores, though the increased load due to stores is included in the network and memory latency calculations.

The memory is modeled as a queue with service time equal to the memory cycle time (see [19]). Although the actual hardware does not queue requests at the memory, but rather blocks the last stage of the network, the queueing delay at the memory is an approximation of the extra latency due to this blocking. Simulation results show that the overall latency is more sensitive to increases in service rate than to changes in access time. With a heavy load, blocking at the memory can cause severe degradation in performance (see Figure 8).

The program was used to evaluate changes in cache parameters and the effect of variations in memory cycle time. Cache hit ratios naturally have the most significant effect; within

Access time	Cycle time			
	2	4	6	8
2	27	57	84	108
4		59	85	110
6			87	112
8				114

Figure 8. Latency (in network cycles) vs. memory access and cycle times: simulation of 64 PEs, 2×2 switches, 100% offered load.

reasonable ranges of cache hit ratios, processor utilization is acceptably high at expected network loads. Increasing the speed of the processor and network in relation to memory (that is, increasing the memory cycle time in network cycles) has a diminishing effect on performance. As the network load approaches the service rate of the memory, delay at memory causes the processor utilization to decrease, canceling out some of the effect of increased processor speed (see Figure 9).

5. Processing Element (PE) Design Issues

The presence of a PE-MM interconnection network providing a high-bandwidth, high-latency path to memory significantly impacts PE design. The best way to reduce the effect of the network's relatively high latency on PE performance is to increase the time between the issuance of a request and when the data is needed. This is, in general, not easy and is especially difficult given a processor designed around short memory latency. Although only a small number of outstanding requests are required to increase network utilization, additional outstanding requests are useful in reducing the impact of high latency and increasing the achievable processor utilization: the more outstanding requests allowed, the better the performance.

There are also a number of other areas in which PE design in an HPSMM environment is different from that of a conventional uniprocessor. This section discusses ways to allow a PE to have multiple simultaneous requests outstanding on the network, cache design issues, ways of reducing periodic interrupts, and methods of ensuring cache coherence. We plan to implement some of these ideas in the Ultra III prototype.

5.1 Multiple Processors on a Network Port

For a given size network, system performance can be optimized by having more than one processor share a single network port (as in the original design of the IBM RP3 [23]). This is a good optimization for large systems since with $O(n)$ processors, the network is of size $O(n \log n)$. Sharing a port multiplies the number of outstanding requests by the number of processors sharing it and hence increases the effective network bandwidth. However, the higher bandwidth results in higher latency and worse PE utilization; there is a tradeoff between network and PE utilization.

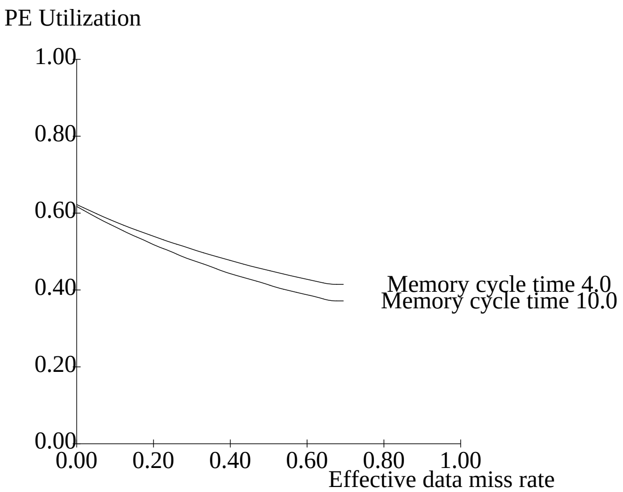

Figure 9. Processor utilization with an effective instruction miss rate of 2%

An alternative is to have a processor that rapidly switches between multiple execution streams when a request is submitted to the network, essentially the approach taken by the Denelcor HEP [4] and now the Horizon and Tera computers [28]. Fundamentally, there is no difference between this and the previous approach: Multiple processors per port are emulated in the same manner that the peripheral processors of the CDC 6000 Series machines [2] share common logic. Whether the whole processor is duplicated (RP3) or just registers and state (HEP, etc.), utilization of processor components is being traded against network utilization. However, to the extent that components are shared among the ‘‘processors’’, component utilization is improved by such a strategy.

5.2 Prefetch Strategies

A conventional processor operates more efficiently in a high-bandwidth, high-latency environment if as large an interval as possible is provided between the times when a memory request is initiated and when the result is needed. Accomplishing this is very difficult; nevertheless, there are several ways to move in this direction.

We consider instructions and data separately. Instructions are read-only and can be safely cached. For a sufficiently large cache and large-grained parallelism, once the code is initially loaded, the cache acts as a local memory, providing instructions without network requests. With small-grained parallelism, aggressive prefetching strategies would be a significant help. One approach is to have a separate instruction prefetch unit similar to that used in the CRISP microprocessor [9]. A more powerful extension to this is discussed in [20], however, the cost of performing possibly unnecessary prefetch operations is not as high with a network as assumed in that paper.

Tolerating high latency on shared data fetches, a much more difficult problem, usually requires sophisticated compiler technology (possibly using ‘‘trace scheduling’’ [11]) and ultimately depends on underlying data dependencies in the application. Subject to these limitations, the following techniques seem promising:

(1) The compiler positions the load of a datum as far as possible in front of its use. Hardware interlocks are provided on registers. For a load, the processor marks the register ‘‘unavailable’’ and transmits the address and register number on its bus. A port is provided to directly write registers; it is used to complete the load and mark the register ‘‘available’’. Until the data has been copied to the register, references to it result in a pipeline stall[2]. This method has the advantage that no additional instructions are needed.

(2) Another approach provides an explicit ‘‘prefetch’’ operation so that a processor can inform the network that it will likely need a specific shared-memory location in the near future. Because of the network’s high bandwidth, this operation is efficient even if the data is sometimes not used. Examples of this method are to prefetch data in the next iterate of a loop with a known stride and along both directions of an upcoming conditional branch. This method has the disadvantage that extra instructions are inserted into the code stream but its advisory nature allows it to be effective in the absence of sophisticated compiler techniques.

[2] The Am29000 provides part of this support. It allows execution to proceed after a load instruction is initiated, while waiting for the load to be satisfied. However, no additional loads or stores may be performed until the data has been provided to the processor, although the processor will provide the network with the address of the next load in some cases.

5.3 Using Existing Microprocessors

Most existing microprocessors require an acknowledgement to a previous request before issuing another, although some allow one instruction and one data reference to be simultaneously outstanding. Nevertheless, there are ways of improving the number of outstanding requests for such processors. For example, if the request is for an instruction word, subsequent words are likely to be needed and multiple network requests can be issued in response to the processor request; the additional words are stored until needed. In addition, the design of the network interface for the AMD Am29000, described below, uses novel features of the Am29062 cache controller to produce additional outstanding requests.

Finally, on all processors, stores can be pipelined: when the processor issues a store, an acknowledgement is immediately generated and the store is sent to the network. This is similar to the use of ‘‘write buffers’’ in conventional processor design.

5.4 HPSMM Caching Strategies

Caches in a shared-memory multiprocessor act as local memory that can be dynamically reloaded. A large I-cache is essential; large D-caches are less effective because of the smaller amount of cachable data in many applications. We present some features that would be present in an ideal cache for HPSMMs.

The amount of shared information (e.g., the instruction stream) is much higher on a multiprocessor than on a uniprocessor and the size of a task may be smaller; this increases the probability that one process can use data left in a cache by another process. In addition, since the cost of reloading invalidated data is much more expensive in such a system, it would be advantageous to have context switches occur without evicting data from a previous task. To allow the reuse of data left in the cache, a unique segment identifier associated with every physical memory segment can be stored in the cache tag. The cache can be designed to invalidate any line containing a word of a specified segment, given the segment ID. The operating system prevents the use of stale data by maintaining a generation count for each segment in the system. This count is incremented when a PE modifies the segment. When a process is scheduled, if one of its segments has a generation count different from that last used for the segment by the PE, all cache lines of that segment are removed from the PE’s cache. This is the only time cache flushes are needed.

The choice of write-back vs. write-through caching strategies often depends on the type of data being cached [19]. Often, fewer memory requests are generated by a write-back cache policy for the extremely volatile stack, though this effect will be less prevalent on RISC processors such as the Am29000 that support register windows. However, in a network-based system we are not as concerned with reducing memory traffic rates as we are in distributing the traffic over time and avoiding ‘‘bursts’’ of traffic that can saturate the network and increase latency for other PEs. With a write-back policy, when a cache miss occurs and a dirty line must be evicted, the PE must issue both a store of the old data and a load of the new data. Also, when a cache flush occurs, a large burst of traffic is generated. In network environments, these factors produce disadvantages for write-back caches that may outweigh their advantages. Over the next year, we plan to couple a trace-driven parallel-programming simulator currently used for studies on multiprocessor paging [29] with our network simulator. This new simulator should allow us to evaluate these effects.

5.5 Reducing the Cost of Periodic Interrupts

Periodic clock interrupts, conventional in uniprocessor systems, have a high overhead in multiprocessor systems. Clock interrupt code occupies space in the cache and TLB and may increase their miss rates. Also, the bursts of requests generated by all processors simultaneously servicing a clock interrupt, though somewhat reduced by combining, tend to saturate the network.

Clock interrupts are used to maintain the time and provide interval timing (such as CPU quantum expiration). In the absence of explicit interprocessor interrupts, multiprocessor operating systems also use clock interrupts to periodically check for events, such as a process being added to the ready queue or a signal being delivered to a running process. In order to eliminate the need for these periodic interrupts, we discuss alternate methods of providing the above functionality.

For timing functions, a chip that provides a variety of clocking and counting operations, such as the Am9513 System Timer, is used in the Ultra III design. It maintains the time and performs interval timing; interrupts are only required when an interval expires.

To support the notification of events, two interprocessor interrupt mechanisms are provided. The first, known as *reflection* and described in [10], communicates with the PE running a specific process, to deliver a signal to that process. Each time a process is run by a PE, the PE number is placed in a known location, called a *reflector*, associated with that process; when it is not being run by any PE, the reflector is set to an undefined value. To notify the appropriate PE, a reflect operation is directed at a reflector. This operation is received by the MM containing the reflector, which reads its value and returns it to the originating PE. If a PE is running the process, the MM also sends a message that interrupts the PE and causes it to deliver the signal to the process. If no PE is running the process, the requestor is notified and handles delivery itself. This mechanism allows the signal to be delivered to a process that is free to migrate among PEs. However, additional higher-level synchronization is required for the case where the target process is picked up by another PE before the requesting process receives its response. The use of the network for the reflection operation has useful properties; see [10] for details.

For other events, a number of bused interrupt lines are provided. Each line is received by all PEs and can be driven by any of them or by an external processor. Using mask bits, PE's selectively enable which lines may interrupt them. This creates groups of PEs to be interrupted for various events. For example, one line could interrupt all idle PEs, another all PEs running processes at lower than a certain priority, etc. Along with the use of the timer chip and reflection, this substantially reduces the number of unnecessary interrupts and allows more rapid responses to events. Facilities similar to these are provided to enable user-mode tasks to receive and send signals without having the kernel service a real interrupt.

It should be noted that this feature can also be used to provide an efficient implementation of the user-mode ''barrier'' primitive. The conventional implementation continually polls a shared location to see if all tasks have reached the barrier. By reserving an interrupt line to be asserted by the last task to reach a barrier, not necessarily the barrier of interest, we may poll less frequently by introducing a delay between successive polls. The assertion of this line causes the polling PE to make an immediate poll, rather than waiting for its delay time to expire. This implementation does not require the bused interrupt lines to be reliable; missed or extra interrupts will affect performance if they occur at an excessive rate, but will not affect correct execution.

5.6 Maintaining Data Consistency

If two PEs have a shared read-write variable in their caches, a modification by one PE is not seen by the other. This is the well-known problem of cache consistency [27] to which there is no known general solution in systems where the caches cannot be interconnected. We therefore prohibit the caching of shared read-write data unless the program specifically permits it and takes responsibility for maintaining cache consistency or certifies that it is unnecessary.

However, there is an additional potential for data inconsistency because stores are acknowledged before memory is modified and the network does not preserve FIFO ordering; see [26] for a complete discussion of this issue. As an example, suppose two tasks, **A** and **B**, are sharing two variables, X and Y, initially zero and execute the following:

Task **A**	Task **B**
X := 1; Y := 1;	MyY := Y; MyX := X;

Although we cannot predict the values of MyX and MyY, the case that MyX = 0 and MyY = 1 is incorrect. To prevent such behavior in a multiprocessor system, the *fence* operation is supplied by the PE board. This operation is also used by the RP3 [23] and forces a PE to wait until all requests issued to the network are acknowledged. Fences are required in both tasks **A** and **B** above; [26] discusses the general case of when they are required.

Most fence operations need only wait for the acknowledgement of stores of noncachable data since cachable data is not considered shared read-write data for the reasons discussed above. However, when a context switch occurs, it must be ensured that all modified data has been written to memory before another PE picks up the process. Two fence operations, NFENCE and AFENCE, are defined and implemented by the PE; the former waits for only the noncachable requests and the latter waits for all requests. Since current compilers do not insert the required NFENCE operations, an ‘‘implicit fence’’ mode is provided. When it is enabled, an NFENCE is performed before each noncachable operation.

6. PE Design Examples

In this section, we present an overview of two PE implementations. One is for a CISC processor, the Motorola MC68030, and the other is for the AMD Am29000, a RISC processor. Both implementations are presented to provide two examples of how to adapt conventional, existing microprocessors for use in a HPSMM system. The Am29000 is used in the Ultra III prototype design for higher performance and easier interfacing.

6.1 Motorola MC68030 PE Board Design

The structure of this PE board is shown in Figure 10. The 68030 has separate on-chip I- and D-caches, each of which has a 128-bit line size. Its bus requires two clock cycles to transfer 32-bits. To fill a cache line, the 68030 requests a ‘‘burst mode’’ bus cycle. If the external interface complies with the burst, 128 bits of data are transferred in five clock cycles. Otherwise, 32 bits are transferred in two cycles and the 68030 does not request the remainder of the line until the data is actually needed. In the case of the I-cache, burst mode provides important prefetch capability and substantially increases the hit rate of the on-chip cache. To provide the burst response to the on-chip cache, the off-chip I-cache must have 128-bit lines.

On an off-chip I-cache miss, it is advantageous to obtain all 128 bits at once. This eliminates the complexity of multiple validity bits per line and increases the amount of prefetch. Either one 128-bit request (assuming that memory is interleaved appropriately) or two 64-bit requests (to different MMs) can be issued. To decide between the two options, note that a single 128-bit request has a six-packet response and that network latency increases with message length. On the other hand, two 64-bit requests require waiting for both responses, but utilize two, rather than one, memory modules. Network simulations show that these two approaches are close in performance; we have chosen to use two 64-bit requests to avoid supporting six-packet messages in the network.

It is not as important to provide the 68030’s D-cache with 128 bits in a single burst. Because of the lower locality of reference, the increase in the hit rate of the on-chip D-cache due to prefetching is less than that for the I-cache. Also, the processor would wait for data that it may not use. Finally, because the data cache is writable by the processor, additional logic would be needed to maintain the multiple validity bits per cache line required to support write-allocate.

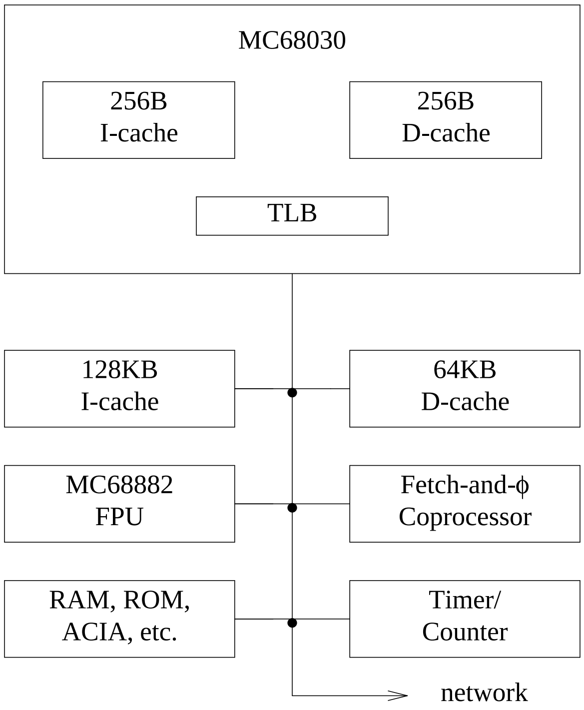

Figure 10. Structure of the MC68030 PE Board

Therefore, our off-chip D-cache has a 32-bit line size with a single validity bit per line (we do not write-allocate for partial-word stores).

The interface between the 68030 and the network is a state machine with states for the various operations such as noncachable load, I-cache hit, D-cache hit, and store. It stalls when no ORIs are available, when required by a fence operation, or when the network port to memory is blocked.

Only 3 ORI bits are used in this design. By reserving the ORIs of 0 and 1 for loads (and fetch-and-ϕs) and using 2-7 for stores, the PE can determine if a response is for a load or store by merely looking at its ORI. For ORIs of 0 and 1, the type of operation is determined from the PE's state machine.

Two bits of data are maintained for each of the six ORIs reserved for stores. One indicates whether or not a pending store is associated with that ORI; if so, the second indicates whether or not the request is to a noncachable location. Stores must be delayed when all ORIs are in use. The AFENCE operation waits until all ORI bits are available and NFENCE waits until all ORI bits used for noncachable stores are available. ORIs are made available by monitoring the responses from the network; this is the only processing required for store responses.

The 68030 instruction set must be augmented to include the fence, reflection, and fetch-and-ϕ operations either by using the 68030 coprocessor interface or memory-mapped registers. Because the latter method requires user-mode mappings for these registers and because of the scarcity of TLB entries in the 68030, the coprocessor interface is used. To issue bus cycles for the reflection and fetch-and-ϕ operations, the physical address is needed. Since the MMU is located on the 68030 chip, there is no direct way to obtain the physical address; instead, the coprocessor requests the data from the 68030, detects which bus cycle is actually being used to obtain it, and issues the special operation to that address Special logic must be added to handle the case of the data being in the on-chip cache.

Finally, the PE provides a connector for an auxiliary board containing an ACIA, RAM, and ROM to be used for diagnostic functions.

6.2 AMD Am29000 PE Board Design

The structure of this PE is shown in Figure 11. It uses the AMD Am29000 CPU, a RISC microprocessor that performs at 17 MIPS when run at 25MHz; the Am29027 floating-point coprocessor that performs at 3.1 LINPACK MFLOPS (single-precision); and the Am29062 [3], a sophisticated cache controller containing 8KB of data in 512 128-bit lines with two-way set associative organization. The Am29062 can be programmed for various sub-block sizes, prefetch and write policies, and either write-through or write-back caching policies, in addition to many other options. In this section, we discuss the design of this PE board, concentrating on those areas specific to a HPSMM system. This design should be contrasted to that of the MC68030 discussed above.

The Am29000 CPU has three 32-bit buses: address, instruction, and data. Addresses need not be presented during the entire transaction, so instruction and data references can proceed simultaneously. It also supports a ‘‘burst mode’’ where a single address is presented and subsequent accesses reference consecutive words. In addition, the address of the next request can be placed on the address bus for previewing while a previous request is being processed; this is called a ‘‘pipelined’’ request. A device can use the ‘‘preview’’ to speed up response time when the request is actually made. Thus, unlike the 68030, up to three simultaneous load requests can be issued by the 29000: one primary request on each port (instruction and data) and one pipelined request on one of the ports. In addition, a network interface can prefetch additional words in advance of their being requested for burst requests. Since, when no fence is present, stores can be acknowledged once they are transmitted on the network, a substantial number of simultaneously

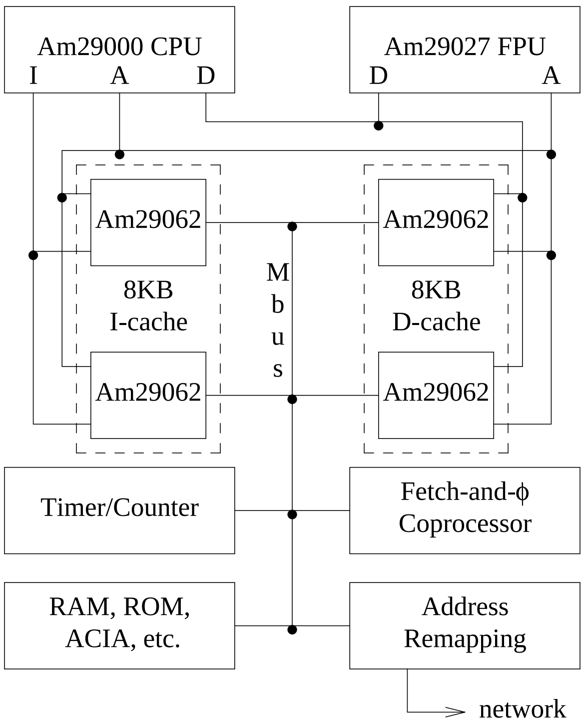

Figure 11. Structure of the Am29000 PE Board

outstanding requests from a single PE can be in network transit.

In addition to being able to issue multiple simultaneous loads, the 29000 has features that minimize the effects of long memory latency for both instruction and data fetches. It prefetches instructions into a 4-word buffer, thus essentially pipelining instruction fetches. Therefore, an instruction word is usually not required until four cycles after it is requested. Although this is small in comparison to the network latency, it does help. For data references, the 29000 simply marks the destination register unavailable and allows execution to proceed until either the register is referenced or another load is issued. In the latter case, the address is placed on the bus as a pipelined request.

In the Ultra III design, however, the network is not directly connected to the 29000, but rather to the caches, so a different class of requests are simultaneously outstanding. The 29062s are connected directly to the address, instruction, and data buses of the 29000 and are connected to a common ''memory bus'' which, in our case, is a network port. This bus multiplexes address and data lines and supports simultaneous read requests from multiple cache chips sharing the bus (write requests are acknowledged without waiting for the network response). Single-word load requests are issued as fetch-and-adds with zero increment and burst requests produce two load double-word requests; the returned words are saved in a dual-ported RAM until requested by the 29062. Two load requests may be outstanding from each of 4 Am29062s (two each for instruction and data), in addition to any pending stores, though it is doubtful whether any pattern of references from a single PE could generate all possible load requests at the same time.

There is one case where the presence of the cache may cause fewer requests to be simultaneously outstanding. When two shared (and hence usually noncachable) loads are made in such a manner that the 29000 can pipeline these loads by placing the address of the second load on its address bus while the first load is outstanding, the cache controller does not process the second load until the first is completed. To be able to issue this request as soon as the processor presents its address, a special interface must be placed ''around'' the cache to perform this function. A dedicated ORI is reserved for requests made in this case.

One thing to keep in mind is that there is a small chance that data requested from the network will actually not be requested by the caches, since ''burst mode'' prefetches may be cancelled. If the probability of needing the data is sufficiently high, the added traffic is justified to lower the latency when the request is eventually made. The ''break-even'' probability is a function of many system parameters; a study needs to be done to quantify the break-even point.

ORI processing for this PE requires 4 bits per ORI and is similar to that of the 68030. Handling of ORIs for stores is identical; ORIs for loads are assigned depending on which 29062 makes the request and whether it is the first or second request issued for a burst. Up to 9 outstanding loads and 7 pending stores can be supported.

The remaining issue relates to adding ''instructions'' to the processor to support cache operations, fence, reflection, and fetch-and-ϕ operations. Unlike the 68030, the coprocessor interface of the 29000 is designed for only a single coprocessor, in our case the 29027 floating-point unit. However, the 29000 has a second distinct address space, the I/O address space, which can be used, at the expense of a TLB entry, to perform these operations. Cache operations can be conveniently done by exploiting a feature of the 29062 that allows operations to be initiated from the memory bus.

Fetch-and-ϕ operations have the same problem as on the 68030: The user must supply a physical address but there is no way to perform a virtual-to-physical mapping off-chip. There is, however, a simple solution in the case of the 29000: Each load and store instruction has an additional bit (called the *lock* bit) that can be set by the user, has no internal effect, but causes the 29000 to present this bit externally when the request is issued. We use this bit to flag fetch-and-ϕ operations. In such a case, the operation type and operand must previously have been sent by an I/O-space store. Then, when the flagged load request is issued, the address is translated to a physical address by the normal operation of the load instruction. Presenting a transaction to the 29062 with the *lock* bit set causes it to invalidate the specified address if it is present; this is exactly the behavior we require for fetch-and-ϕ operations.

Finally, like the 68030, the PE can provide an ACIA, ROM, and on-board RAM.

7. Conclusion

We have discussed problems in PE design unique to highly-parallel, shared-memory multiprocessor systems using a high-bandwidth, high-latency interconnection network to connect processing elements to memory modules. An analysis of such a network was presented along with its implications on system design. Most of the issues presented apply to any system using a similar network. Two designs of PEs, one using a CISC processor and one using a RISC proessor, were presented.

A 64-PE Ultra III prototype is planned using Am29000 processors; with these chips we expect to achieve an aggregate performance of 500 MIPS and 60 MFLOPS. At present, we expect this prototype to be functional in mid-1990, with a four-PE pre-prototype functioning a year earlier. When Ultra III has been completed, it will allow analysis of processor and network behavior, such as cache hit ratios and number of combines, and also be a good vehicle for testing and evaluating various parallel algorithms. We expect to report on such data when it is available.

Acknowledgements

Much of the work described was done by various members of the NYU Ultracomputer Project. Specifically, Allan Gottlieb wrote the first network simulator and Yue-Sheng Liu did much of the network simulations and analyses. Many people in the Ultracomputer group were involved in the design of Ultra III. Other than the authors, these include Ron Bianchini, Jan Edler, Allan Gottlieb, Mal Kalos, and Jim Lipkis; packaging and detailed design were done by Ron Bianchini, Jeff Eichel, and Jai-Rui Wang. The IBM RP3 project also provided useful ideas.

References

[1] A. Borodin and J. E. Hopcroft, ‘‘Routing, Merging, and Sorting on Parallel Models of Computation’’, *Proc. of the 14th Annual ACM Symp. on Theory of Comp.*, pp. 338-344, 1982.

[2] Control Data Corporation, *Control Data® 6400/6500/6600 Computer System, Reference Manual,* 1965.

[3] B. C. Cole, ‘‘AMD Puts Cache Memory on One Programmable Chip’’, *Electronics,* pp. 87-88, April 28, 1988.

[4] Denelcor, *Heterogeneous Element Processor, Principles of Operation*, 1981.

[5] S. Dickey, A. Gottlieb, and R. Kenner, ‘‘Using VLSI to Reduce Serialization and Memory Traffic in Shared Memory Parallel Computers’’, *Advanced Research in VLSI: Proc. of the Fourth MIT Conf. on VLSI*, C. E. Leiserson (ed.), pp. 299-316, 1986.

[6] S. Dickey, A. Gottlieb, R. Kenner, and Y. Liu, ‘‘Designing VLSI Network Nodes to Reduce Memory Traffic in a Shared Memory Parallel Computer’’, *Circuits, Systems, and Signal Processing*, Vol. 6, No. 2, pp. 217-238, 1987.

[7] S. Dickey, R. Kenner, M. Snir, and J. Solworth, ‘‘A VLSI Combining Network for the NYU Ultracomputer’’, *Proc. of the IEEE Intl. Conf. on Comp. Design,* pp. 110-113, 1985.

[8] S. Dickey and Y. Liu, ‘‘Simulation and Analysis of Different Switch Architectures for Interconnection Networks in MIMD Shared Memory Machines’’, NYU Ultracomputer Note No. 141, June 1988.

[9] D. D. Ditzel and H. R. McLellan, ‘‘Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero’’, *Proc. of the 14h Annual Sym. on Comp. Arch.*, pp. 2-9, 1987.

[10] J. Edler, A. Gottlieb, C. Kruskal, K. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, ''Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach'', *Proc. of the 12th Annual Intl. Conf. on Comp. Arch.*, pp. 126-135, 1985.

[11] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, Cambridge, Ma., 1986.

[12] M. Franklin and S. Dhar, ''Interconnection Networks: Physical Design and Performance Analysis'', *Journal of Parallel and Distributed Computing*, Vol. 3, pp. 352-372, 1986.

[13] L. R. Goke and G. J. Lipovski, ''Banyan Networks for Partitioning Multiprocessor Systems'', *Proc. of the First Ann. Symp. on Comp. Arch.*, pp. 21-28, 1973.

[14] A. Gottlieb, ''An Overview of the NYU Ultracomputer Project'', *Experimental Parallel Computing Architectures*, J. Dongarra, ed., North Holland, pp. 25-95, 1987.

[15] A. Gottlieb, B. Lubachevsky, and L. Rudolph, ''Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors'', *ACM TOPLAS*, Vol. 5, No. 2, pp. 164-189, Apr. 1983.

[16] C. Kruskal and M. Snir, ''The Performance of Multistage Interconnection Networks for Multiprocessors'', *IEEE Trans. on Computing*, Vol. C-32, No. 12, pp. 1091-1098, Dec. 1983.

[17] C. Kruskal, M. Snir, and A. Weiss, ''The Distribution of Waiting Time in Clocked Multistage Interconnection Networks'', *Proc. of the 1986 Intl. Conf. on Parallel Processing*, pp. 12-19.

[18] D. Lawrie, ''Access and Alignment of Data in an Array Processor'', *IEEE Trans. on Comp.* Vol. C-24, No. 12, pp. 1145-1155, Dec. 1975.

[19] K. McAuliffe, *Analysis of Cache Memories in Highly Parallel Systems*, Ph.D. Thesis, Courant Insitute, NYU, May 1986.

[20] H. R. McLellan, *Instruction Prefetch Strategies in a Pipelined Processor*, Master of Science Thesis, MIT, Feb. 1983.

[21] Motorola, *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, 1987.

[22] A. Norton and G. Pfister, ''A Methodology for Predicting Multiprocessor Performance '', *Proc. of the 1985 Intl. Conf. on Parallel Processing*, pp. 772-781.

[23] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss, ''The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture'', *Proc. of the 1985 Intl. Conf. on Parallel Processing*, pp. 764-771.

[24] G. Pfister and A. Norton, '''Hot Spot' Contention and Combining in Multistage Interconnect Networks'', *IEEE Trans. on Comp.*, Vol. C-34, No. 10, pp. 943-948, Oct. 1985.

[25] J. T. Schwartz, ''Ultracomputers'', *ACM TOPLAS*, Vol. 2, pp. 484-521, 1980.

[26] D. Shasha and M. Snir, ''Efficient and Correct Execution of Parallel Programs that Share Memory'', NYU Ultracomputer Note No. 96, Mar. 1986.

[27] A. J. Smith, ''Cache Memories'', *Computing Surveys*, Vol. 14, No. 3, pp. 473-530, Sept. 1982.

[28] B. J. Smith, Private Communications, 1988.

[29] P. J. Teller and R. Kenner, ''Evaluation of TLB Consistency Techniques for Highly-Parallel Shared-Memory Multiprocessors'', NYU Ultracomputer Note, in preparation.

[30] J. Wilson, *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add*, Ph.D. Thesis, Courant Institute, NYU, June 1988.