ULTRA III: IMPLEMENTING A SCALABLE SHARED-MEMORY MULTIPROCESSOR

Summary

Parallel computers are necessary to achieve the much faster aggregate speeds needed in the future. Although uniprocessor performance is still increasing, expensive, exotic, and power-hungry technologies are required to attain current and future supercomputer-level performance. In addition, both physical limits, such as the speed of light, and quantum effects become significant as uniprocessor performance increases.

The Ultracomputer Research Laboratory has pioneered in the design of shared-memory MIMD systems, one of the major promising classes of parallel computing architectures. The Ultracomputer architecture connects thousands of processing elements, each containing a cache, to an equal number of memory modules via an interconnection network constructed of custom VLSI components that combine (merge) requests from different processing elements destined to the same memory location.

This proposal begins a program to construct Ultracomputers with one or four thousand processors. A modest 16-processor engineering prototype will be constructed by NYU in cooperation with AMD, one of its industrial partners. In the second stage, MCC, NYU's other industrial partner, will construct a 256-processor prototype. The latter prototype will be large enough to assess and experiment with the Ultracomputer architecture. Specifically, we expect to demonstrate the feasibility of constructing a network that combines memory requests. This ''proof-of-concept'' will encourage commercial exploitation of shared-memory, parallel-processing systems.

1. Overview

Most highly-parallel processing systems to date have focused on private-memory, message-passing designs. The only commercial shared-memory MIMD system currently available, beyond bus-connected systems with modest numbers of processors, is the BBN Butterfly. This relative scarcity of highly-parallel shared-memory machines is due to a common perception that they are difficult to implement efficiently. However, shared-memory machines form a promising class of parallel computing architectures with important applications for which its programming model is preferable. The present proposal addresses the design and implementation of such systems.

1.1. Background of the Ultracomputer Architecture

The NYU Ultracomputer project has studied shared-memory architectures throughout the decade. Successful use of highly-parallel, shared-memory MIMD systems requires avoiding serial bottlenecks at all system levels, from algorithm design through hardware components, and thus providing *scalability* as the number of processors grows. The nature of serial bottlenecks in applications, programming environments, operating systems, coordination primitives, system architecture, and within processor-to-memory interconnection networks has been researched extensively by our group. We have developed techniques either to eliminate such bottlenecks or to reduce their impact significantly.

Numerous bottleneck-free coordination primitives have been designed by our group and implemented in both a highly parallelized UNIX system testbed and libraries for application programs. Our current operating system, which we now call Symunix I, is a version of UNIX V7 whose kernel uses our new coordination primitives to provide a nearly bottleneck-free operating system. We are implementing a new kernel which will support both Mach and Berkeley 4.3 BSD system calls. This kernel will also be nearly bottleneck-free and will serve as a testbed for new methods of user-mode control over highly-parallel tasks with minimal kernel-mode intervention. Although the present proposal does not request funding for operating system development, we intend to port this system to the Ultra III prototypes.

We have constructed a series of prototypes; two eight-processor prototypes have been running our parallelized UNIX system in production use for over four years. At the applications level, numerical analysts and computational physicists have researched the highly parallel implementation of important algorithms. Our varied experience with the eight-processor prototypes built so far has proven invaluable from every point of view. The prototypes have been an indispensable tool for the installation, debugging, and thorough testing of our parallel operating system, compilers, and other software. We and others have been able to experiment with porting existing application programs and writing new ones in the wide variety of styles that the Ultracomputer supports. Not least, it has given us confidence in our ability to design, debug, test, and bring into reliable performance the next prototype. It is noteworthy that for the last few years, the mean time to failure of the hardware and software of our prototypes has been on the order of weeks.

A scalable processor-to-memory interconnection network must provide bandwidth proportional to the number of processors. This requires buffered, pipelined networks. Extensive simulation studies and mathematical analyses of the network have been conducted to obtain accurate models predicting the behavior of the network under various conditions. In addition, we have studied methods of physically packaging large omega networks and are confident that systems with thousands of processors are buildable.

Our major contribution to network research involves techniques to eliminate serialization bottlenecks in the network, via *combining* of memory references. As discussed in section 2.3, combining is an important component in systems that scale efficiently to large numbers of processors. Along with performing analysis and simulations of combining, we have developed a semi-systolic VLSI design methodology for the network switches that allows implementation of a variety of combining methods. Our colleagues at IBM (Pfister and Norton [85]) used a network simulator to study the effect of combining memory references and found that for a conventional network, memory ‘‘hot spots’’ are disastrous but, when the network supports combining, the problem disappears. We have constructed sample non-combining and combining[1] components using 3μ nMOS, have run applications on a complete system using non-combining CMOS parts, and are in the final stages of designing 2μ CMOS combining chips. Although the combining components require more silicon than their non-combining counterparts and may be more difficult to package because they cannot be bit-sliced as easily, they require only slightly slower clock speeds.

In contrast to the work discussed above, there are three areas in which we have *not* performed significant research: parallelizing compilers, I/O issues, and fault tolerance. Parallelizing compilers are being studied by at least three major groups: University of Illinois (Kuck *et al.* [81]), Rice University (Allen *et al.* [83]), and IBM (Allen *et al.* [88]); their results will be applicable to our systems. I/O issues, though important, are less critical for our major application interest, scientific computation. Aside from a characterization of general techniques for multiprocessor I/O (Edler *et al.* [85]) and some recent work on the I/O requirements of demand paging (Teller [89]), we have done little in this area.

Finally, providing acceptable availability of large parallel systems requires the use of explicitly fault tolerant techniques. We have not been active in this area, but much work has been done on making omega networks fault tolerant (Feng and Wu [81], Adams and Siegel [82], and Lilienkamp *et al.* [82]). A recent paper by Banerjee and Dugar [89] addresses fault tolerance for omega networks capable of combining fetch-and-add requests. The proposed 256-processor system will have approximately the same component count as current high-end computer systems and hence should achieve the same overall reliability as such systems, without the need for explicitly fault-tolerant designs. However, larger systems, such as a 4096-processor Ultracomputer, will need to utilize the techniques being developed for fault-tolerant networks.

1.2. Proposed Prototypes

From its inception, the goal of the Ultracomputer project has been the design of systems with thousands of processors using technology expected to be available in the 1990s. With 1990 fast approaching, technology

[1] The constructed combining chips, and those proposed for this prototype, only combine pairs of requests; i.e., if three requests for the same location arrive in a buffer simultaneously, one pair will be combined and the third will be transmitted unchanged. Recent studies by Lee *et al.* [86] indicate that for thousands of processors, this ‘‘two-way’’ combining may not be sufficient; it may be necessary to merge three or more incoming requests into a single combined request. A straightforward enhancement to our basic technique can perform this operation (see Liu and Dickey [88] for a simulation study of one possible implementation).

has reached a level where it is now practical to construct such systems at reasonable cost. It is now time to embark on a program whose goal is the construction of an Ultracomputer with 1024 or 4096 processors. A 4096-processor Ultracomputer constructed with present technology would have a size of 15′×9′×3′, dissipate about 100KW, and have a construction cost of about $10 million. This system would have a *sustainable* performance of about 60 GIPS and 10 GFLOPS.

The program begins with this proposal. During its first year, the design of the three system components (processing element, memory module, and interconnection network node) will be completed (preliminary versions of each are now working in our lab), and, in cooperation with AMD (Advanced Micro Devices), a 16-processor engineering pre-prototype will be constructed. During the second year the experiments described below will be performed on this system. Meanwhile, a second proposal, most likely to DARPA, will call for our other industrial partner, MCC (the Microelectronics and Computer Technology Corporation), to construct a 256-processor Ultracomputer using the design validated by the 16-processor system but repackaged with advanced ‘‘no-backplane’’ technology described in section 3.4.2. The present proposal includes funding for NYU assistance with this construction, which is to begin when the pre-prototype has been validated. Commitment letters from both our industrial partners are attached.

While there is much support for the theoretical concepts of the Ultracomputer, especially the usefulness of fetch-and-add and the idea of combining, there remains some healthy skepticism that a high performance combining chip can be built as we propose, and that a fully functional network can be assembled and used effectively in large systems. Building a prototype with support for multiple outstanding memory references (see section 2.2), a large number of modern processors, and a combining network is the only convincing way to answer crucial questions about efficiency and practicability of the architecture and to ensure that no design holes remain.

The 16-processor and especially the 256-processor prototypes will be well suited for a variety of experiments. The large prototype will be one of the fastest shared-memory scalar MIMD machines in the world, will have sufficient memory to run many applications of scientific interest in their own right, and will permit comparisons between using few and many processors. Studies of system performance, especially speedups, on these applications are an important step toward assessing the effectiveness of systems consisting of a few thousand processors.

As shown by Pfister and Norton [85], the percentage of requests that must be directed at a single location before serious performance degradation is observed is roughly inversely proportional to the number of processors. This percentage also marks the onset of significant combining, which eliminates this particular performance degradation. With 16 or fewer processors, significant combining would only occur for contrived examples, for example processes that repeatedly coordinate with no useful intervening activities. With the 256-processor prototype, combining will be observed even with nontrivial computation between coordinations. (Note that hot-spots in the memory will be accentuated by the caches, which filter out many references to non-shared data.) These prototypes will allow us to quantify both the cost and benefits of combining. The VLSI implementation itself demonstrates feasibility; throughput and latency measurements quantify its performance; and the benefits of combining will be observable since the chips will permit combining to be disabled.

Architecture and performance evaluations, research on scheduling techniques, and optimization studies of operating system characteristics will be performed on these prototypes. Other types of experiments, made possible by programmable logic on the processing element, will allow further architectural research to be performed. For example, we will vary the maximum number of outstanding memory requests permitted and thereby quantify the importance of this feature on significant applications. Various methods of distributing requests among the memory modules will also be evaluated. It is likely that many other experiments of this type will be suggested by early studies of the prototype.

The Ultracomputer architecture is described more fully in Gottlieb [87], a copy of which is attached. (Note that its description of software environments corresponds to our views as of 1985. See Edler *et al.* [88a] and [88b] for our current views.) In our architecture, a large number of processing elements are connected, via a high-speed omega network composed of intelligent switching nodes, to an equal number of memory modules (see Figure 1). The processing elements (PEs) are augmented with a group of operations

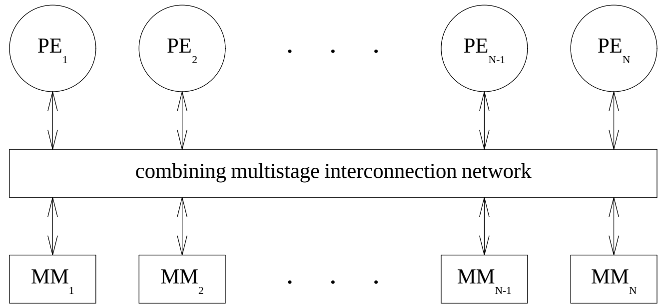

Figure 1: Ultracomputer Architecture

of the form fetch-and-ϕ(X,e), each of which reads a value X and atomically replaces it with ϕ(X,e). (The most important of these operations is fetch-and-add.) The memory modules (MMs) support load, store, and fetch-and-ϕ operations. The interconnection network routes memory requests to the MMs and responses back to the PEs. It is composed of custom CMOS chips that detect multiple concurrent requests for the same memory location and combine them into one request. Section 3 describes the proposed system; see the attached Gottlieb paper for more information, including details on the design of the network and switching nodes.

The design of the PEs is a key part of this research. Section 3.2.1 describes the goals for the PEs. For a number of reasons, discussed more fully in section 3.2.2, we have chosen the Am29000 microprocessor produced by AMD, one of our industrial partners. In addition to being a modern, high-performance RISC processor, it has a number of features making it particularly well suited for our application. The PE (along with the MM and network switch modules) will become part of the ES-Kit family being developed by MCC, our other industrial partner.

1.3. Comparison with Other Shared-Memory Systems

No shared-memory machine containing the components we believe are required for scalable systems has been designed and implemented. Specifically, none have networks that eliminate hot spots caused by process coordination; many are dissimilar in other ways as well. The two systems that are the closest to the proposed system are the BBN Butterfly (Crowther *et al.* [85a] and [85b]) and the IBM RP3 (Pfister *et al.* [85]), discussed below. In addition, an effort to produce a commercial shared-memory MIMD machine is underway at Kendall Square Research Corporation. We are not at liberty to disclose the details of this machine, but can say that it will have a hierarchical memory organization and will not have fetch-and-ϕ instructions.

The current BBN Butterfly does not interleave memory across the MMs, has no caches at the processors, does not allow multiple requests from a single processor to be outstanding in the network, and does not combine requests for the same memory location. Although not officially announced, it is widely known that a successor to the Butterfly is being designed and constructed. This machine will come closer to our design than the original Butterfly but it is our understanding that it will also not support combining and that there are other significant differences between its network and ours.

The goal of the IBM RP3 project was the construction of a 512-processor shared-memory system containing substantially all the components of our architecture, including a combining network. We were closely connected with this exciting project for a number of years and committed considerable resources towards its support, mostly in the areas of operating system development and non-blocking coordination algorithms. Their hardware development has been completed with the construction of a 64-processor system. Unfortunately, this machine is rather disappointing, with lower-than-expected performance and functionality. Specifically, combining has not been implemented[2] and floating-point hardware has not been installed in the processors. The performance of the RP3 is limited both by the use of the slowest version of the IBM ROMP processor running at a slow clock speed and a large latency within the processor and memory interfaces. It is likely that less than 1 MIPS per processor is being achieved, approximately an order of magnitude slower than the proposed prototype.

From our perspective, there are two major reasons why the RP3 fell short of its design goals and appeared two years later than originally planned. The first was that the proper resources were not brought to bear by IBM. The second reason was that the design of the I/O system and processor were unnecessarily complex. This lesson from the RP3 project is one of the reasons that we emphasize simplicity of design in our proposed prototypes.

Although there are several research projects involving hardware combining of fetch-and-ϕ instructions,[3] we have been unable to locate any current or planned commercial effort to produce a large-scale, shared-memory system with this property. We expect the successful completion of the proposed project to spawn such commercial efforts.

2. Scalability

Scalability refers to the extent to which a system approaches perfect (linear) speedup with large numbers of PEs. Fundamental physical laws limit scalability, as discussed in the first subsection below. However, in practical terms, bottlenecks due to serialization points are often the most immediate problem preventing scalability of an architecture; methods for handling these bottlenecks are explained next. Finally, we note the costs of scalable methods and discuss when they are appropriately used.

2.1. Physical Limits to Scalability

Physical laws affect the scalability of *any* architecture. Real processors occupy non-zero volume. The total volume required for a processing system increases at least linearly with the number of processors and may be further increased by other physical limits, such as information density (see Glasser and Zukowski [88]). The restriction that information cannot be transferred faster than the speed of light and must travel across an increasing volume provides a lower bound on communication costs. In machines with a globally-shared, uniformly-accessible memory, this means that memory latency must necessarily grow with machine size. A cache or local memory helps considerably, but unless either the percentage of nonlocal access approaches zero as the machine size increases or compilers can fill an unbounded number of delay slots (both unlikely prospects for most applications), communication delays will necessarily force idle time to increase with machine size and limit scalability.

Latency limitations arise even for machines, such as hypercubes and meshes, where some nonlocal memory is still close. As the number of processors increases, the percentage of total memory that is ''near'' (i.e., within a constant K hops of) a given processor tends to zero unless the degree of the interconnection grows as $\#PE^{1/K}$; such growth is impractical. Hence to achieve bounded average latency with ever larger machines, the number of ''non-near'' references must tend to zero, which again is not likely.

Although perfect scalability cannot be achieved, it is still an important ideal. As Larry Snyder has said, ''We do not live in asymptopia'', and hence achieving high, albeit imperfect, scalability can permit the construction of large, efficient parallel processors. For low-power technologies and shared-memory systems of up to a few thousand processors, memory latency is dominated by fan-in, rather than speed-of-

[2]After an IBM group not familiar with systolic techniques failed to implement a combining switch, the RP3 project contracted with NYU to implement our combining switches on their CMOS process. Plans to include such switches in RP3 were suspended when it became clear that subsequent prototypes would not be constructed.

[3]The BBN Monarch project (Rettberg [87]) has proposed using a combining switch in a bit-serial network; this project is currently inactive. Lipovski and Vaughan [88] describe a novel form of synchronous combining also useful in a bit-serial network; we are not aware of any plans to build a prototype demonstrating this technique. Astronautics Corporation of America has proposed the construction of a system with a small number of high-performance processors using our combining switch technology; this proposal is not expected to lead to a large-scale example of a combining network within the next few years.

light, limitations. Fan-in limitations produce logarithmic growth of memory latency. In systems with thousands of processors latency is typically a few tens of cycles, similar to that seen by today's fastest uniprocessing supercomputers. Nevertheless, dealing with high memory latency is one of the key problems in the design of scalable, shared-memory systems.

2.2. Tolerating Memory Latency

The deleterious effect of significant memory latency may be reduced by exploiting locality of reference to lower the number of requests experiencing such latency. Most multiprocessors provide access to a physically local memory, either separately addressed or organized as a cache; we use the latter. As an extension to this concept, some architectures, e.g., Cedar (Gajski *et al.* [84]) and many nonshared memory architectures, allow certain nonlocal memories to be accessed more rapidly than others. Exploiting this nonuniform access successfully requires an application to be properly structured by either the programmer or system software. We do not use this approach, preferring instead the simplicity of uniformly accessed memory with caches at the processing elements.

Another important technique treats the entire memory subsystem, including the network, as a pipeline. This approach is especially helpful when the memory subsystem provides high bandwidth, as does the buffered and pipelined omega network used in the Ultracomputer, where bandwidth grows linearly with the number of processors. Viewed as a pipeline, multiple memory requests from each processor can be simultaneously present in the network. However, unlike a true pipeline, responses can arrive out of order. Subject only to the sequential consistency constraints discussed in section 3.2.3, many store requests can be outstanding in the network. As described in section 3.2.5, the Am29000-based PE permits several outstanding load requests as well.

Although the AMD processor is a good design for high-bandwidth, high-latency systems such as ours, we will also study methods for further improving the latency-tolerance of PEs. For example, a processor can use register scoreboarding and a split-transaction bus to reduce the effects of latency. Vector instructions can be provided that handle out-of-order arrival of operands. More sophisticated instruction prefetching techniques can also be used.

Besides these modifications to processor design, two other techniques can be used. Multiple processors can be placed at a single PE or the processor can be designed to allow task switching to be performed in one or two clock cycles using methods similar to that used by the Denelcor HEP (Smith [81]). Although these techniques increase the latency-tolerance of the PE, they have the significant disadvantages of increased system cost and they require additional available parallelism. For large systems, combining the use of multiple tasks or processors per PE with intra-processor techniques may produce the best overall performance.

2.3. Serial Bottlenecks and Combining

Scalability of the entire system is determined by the least scalable component. For example, a single system bus, with its fixed bandwidth, becomes saturated as the number of processors grows. In order to scale, such a system would need bus bandwidth increasing linearly with the number of processors.

Another bottleneck occurs when a large number of memory requests are concentrated at a small number of modules. The typical spatial locality in most programs will cause this to occur if consecutive virtual addresses are stored in the same memory module. Using low-order address bits to interleave memory across the modules exploits spatial locality to distribute the references, thus avoiding this potential bottleneck. As in other designs, interleaving is an important part of the Ultracomputer architecture. In addition, the programmable address logic on our processor boards permits experiments using hashing functions to scatter memory accesses uniformly across the modules for more general access patterns.

Interleaving and hashing only help when *different* locations are being accessed by multiple processors. Another potential bottleneck, one that has received considerable attention from the research community, arises from so-called *hot-spot* memory requests (Pfister and Norton [85]), in which a single memory location receives a disproportionate share of memory requests. If these requests are serviced serially, a bottleneck arises. As indicated in detail in section 7.1.3.1 of the attached Gottlieb report, a network with hardware combining can merge concurrent hot-spot requests into a single memory reference, without introducing significant additional delay (cf. section 2.4, below).

To further understand proposed solutions to the hot-spot problem, let us consider a system with an omega network running a parallel application that uses a single shared queue to implement a workpile, where, for simplicity, we assume the queue is never empty and never full.

The most straightforward parallel implementation uses a critical section to protect the insertions and deletions. This critical section produces a bottleneck. More sophisticated queue algorithms, described in Gottlieb *et al.* [83b], use fetch-and-add to eliminate the critical section. Again assuming the queues are not empty and not full, these algorithms execute a constant number of instructions for each queue operation, at least one of which performs a fetch-and-add on a single shared-memory location used for coordination. However, without combining, we have merely succeeded in moving the bottleneck from software to hardware since the fetch-and-adds to the coordination variable are serialized at the memory module. Hence only a fixed number of queue operations could be performed per unit time. Combining eliminates the bottleneck, allowing N processors to complete N queue operations concurrently in the time required for just one such operation.

Yet more sophisticated algorithms using software combining techniques (see Yew *et al.* [86] and Goodman *et al.* [89]), ensure that each memory location is referenced by only a constant number of processors. These algorithms utilize a software tree with each processor assigned to a leaf. Although most processors only ascend a few levels, all must wait for the tree to be fully ascended and subsequently descended, which requires $\log N$ steps in contrast to the constant number of steps in the previous solution with hardware combining. To compare the asymptotic behavior of hardware and software combining, note that the memory access time in any omega-network (with or without combining) grows logarithmically with the number of processors. Hence each queue operation will require $\Theta(logN)$ time with hardware combining and $\Theta(log^2 N)$ time with software combining.[4]

Another way of eliminating the hot-spot is to modify the application to use a non-constant number of queues. However, now load balancing is necessary to prevent some processors from waiting at empty queues when other queues have items. In addition to increasing the difficulty of the algorithm, the load balancing code may require each processor to consult a non-constant number of queues.

Hardware combining increases neither the difficulty nor the computational complexity of the application. We have developed semi-systolic techniques that efficiently implement combining using VLSI CMOS technologies.

The impact of the scalability limitation is dependent on the frequency at which the ‘‘non-scalable’’ operations are performed. For the workpile example, this frequency is determined by the granularity of the work items. Thus the system size at which the techniques discussed in this section are required is application-dependent.

2.4. Costs of Scalable Systems

Our overriding concern for the scalability of key architectural concepts and features is a major distinguishing factor between our project and other shared-memory projects. Our ‘‘design point’’ is systems involving thousands of processors, not 8, 16, or even 64. Accordingly, we make implementation decisions based on the requirements of very large systems, even though these decision may not be the most appropriate choices for smaller systems. For example, if our ultimate goal were a system consisting of 16 processors, it is clear that a bus is more appropriate than an omega network.

Similar issues arise in process coordination software. The bottleneck-free queuing algorithms in Gottlieb *et al.* [83b] have slightly longer path lengths than the straightforward approach using critical

---

[4] The estimate for memory access time holds for systems where fan-in and not time-of-flight limitations are dominant. If the latter dominate, the time required for each queue operation will grow at least as $N^{1/3}$ with hardware combining (since the volume of the system grows at least as N) and $N^{1/3}\log N$ time with software combining. The penalty for software over hardware combining is always a logarithmic factor.

sections. For large systems, probably 64 PEs or greater, the increased path length is justified by the elimination of serialization points. However, for smaller systems, the serialization points are less costly and the more complicated algorithms are not justified.

Another example is our combining network. As discussed above, combining is necessary for systems of sufficiently large size.[5] However, even though we believe, and expect this research will demonstrate, that the cost of combining is not prohibitive, it is not zero either. It is certainly true that for many systems an implementation of the network using 4×4 non-combining switches may well offer a higher performance on many applications than 2×2 combining switches. Nevertheless, we focus on the combining switches because of their importance to the large systems that are our long-term design goal.

3. Ultra III Design

We have followed a conservative path of reflecting on the high-level goals of the Ultracomputer project and studying potential architectures intensively before building any hardware. Analytic modeling and simulation techniques were used to establish the effectiveness of the basic elements of the Ultracomputer design (see for example Snir [81], Gottlieb *et al.* [83a], and Kruskal and Snir [83]). Cache and network simulators (produced jointly with IBM Research) using synthetic data and data derived from real applications were used to estimate the cache hit ratio (McAuliffe [86]) and shared memory latency (Dickey and Liu [88]). The data were then analyzed by a high-level machine simulator produced by IBM to estimate processor utilization (Norton and Pfister [85]). The analytic and synthetic simulation studies plus the application programs examined under simulation by us[6] and by IBM[7] and on the NYU prototypes described in section 4.5[8] indicate that an Ultracomputer is an extremely effective computing engine for a wide variety of programs.

Based on analysis, simulation and our hardware and software experience, we have developed a complete design for a network-based Ultracomputer using current technology. This section presents a concrete description of the Ultra III design, beginning with a description of the interconnection network and including the processing element and memory modules designs. A more detailed description of the Ultracomputer architecture is given in section 2 of the attached Gottlieb [87] report and section 7 of that report presents more detailed hardware designs, albeit slightly dated. Throughout this section, we use the term ‘‘processor’’ to refer to a conventional microprocessor that is part of a processing element (PE). A PE consists of a processor, cache, and an interface to the interconnection network, and possibly other auxiliary devices. Similarly, an MM includes memory chips, control logic, support for fetch-and-phi, and a network interface.

3.1. Interconnection Network

The Ultracomputer network (shown in Figure 2 for 8 PEs) has the topology of an omega network (Lawrie [75]) and the SW Banyan (Goke and Lipovski [73]). It consists of $\log_K N$ stages, where N is the number of PEs in the system and the network is built from $K \times K$ switches. The Ultra III network has $\log_2 N$ stages, each having $N/2$ nodes, where a node is a 2×2 switch that routes memory 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 other switches closer to the PEs and two connecting either to MMs or other switches closer to the MMs.

---

[5] The precise size at which the lack of combining starts becoming a bottleneck is a strong function of many engineering parameters of a specific design as well as the granularity of the parallel tasks and degree of communication in a specific application. Accurately determining the breakeven size for a suite of significant applications would require repeating prohibitively expensive simulations for many choices of engineering parameters.

[6] See Kalos [81]; Rushfield [81]; Korn [81a], [81b], and [81c]; Kalos *et al.* [82]; Lubachevsky [82], Kalos and Zhong [83], Korn and Rushfield [83], and Bishop [83].

[7] See Darema *et al.* [88], Darema-Rogers and Pfister [86], Darema-Rogers *et al.* [87a] and [87b].

[8] See Greenbaum [86], Greenbaum *et al.* [87], and Chiabaut *et al.* [88].

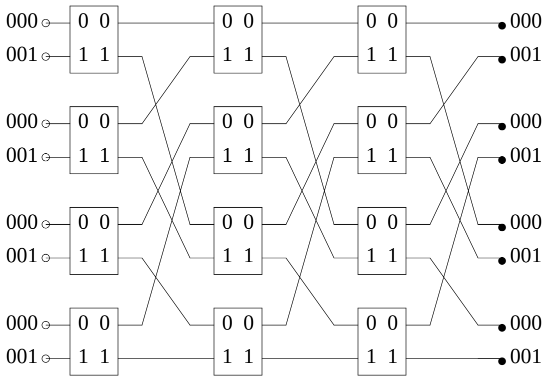

Figure 2: Ultracomputer Network (N = 8)

3.1.1. Switch Structure

A switch consists of two independent routers, one for requests and the other for responses; each contains four queues, one for each input/output pair, and two multiplexors, as shown in Figure 3 (see the attached Gottlieb report for implementation details).

The switch can *combine*, or merge, two requests that refer to the same memory location and that are simultaneously present in its queues. These switches can potentially combine fetch-and-ϕ operations, for any associative operator ϕ,[9] although we implement combining only 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.

One of the main issues in the design of a network is the width of its information path. A 32-bit memory requires a minimum message size of 64 bits (32 for address and 32 for data). Options range from

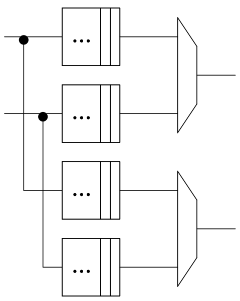

Figure 3: Ultracomputer Switch Structure

[9] *Loads* and *stores* are also fetch-and-ϕ operations. In a load, ϕ is the projection operation $\pi_1(x,y)=x$ so that fetch-and-$\phi(V,e)$ returns the value of V and sets V to $\pi_1(V,e)=V$, i.e. leaves V unchanged. Since e is not used, it need not be transmitted. Similarly, for a store, ϕ is $\pi_2(x,y)=y$ and only an acknowledgement (with no data value) needs to be returned.

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 (Dickey *et al.* [85]) requires that a message inside the queue be composed of an even number of packets and the combining logic is simplified if the address and operation code, needed to decide whether two requests can be combined, are contained in the first packet of the message. The second requirement imposes a minimum width, the first makes a larger width of limited utility. Therefore, 64-bit messages are transmitted as two packets with the first containing the address and the second the data. Less frequent 96-bit load (resp. store) double requests require four packets (one null) in the return (resp. forward) direction.

Messages are pipelined through the network: switches transmit each packet of a message as soon as possible rather than waiting for the entire message. The minimum 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.

3.1.2. 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 could be used for this purpose. However, a major consideration would then be that the PE must be prevented from issuing multiple outstanding requests to the same location.

An alternative method, which we employ and was devised by Pfister *et al* [85], augments each outstanding request for a PE with a unique value, called an *outstanding request index* (ORI); thus the processor must track only 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.

Only the number of the destination memory module needs to be transmitted; the number of the source processor can be computed as the message traverses the network. This is done by having each switch shift the MM number left one bit and insert the incoming port number into the low-order bit. When the message arrives at its destination, its MM number field will contain the PE number. In addition to reducing the amount of information transmitted, this method also allows each switch to use the most-significant bit for routing.

For a 256-PE system supporting sixteen distinct operation codes, sixteen outstanding requests per PE, and an address space of 1GB, the format of the address packet used in requests to memory is the following:

operation code	parity	MM number	ORI	word address in MM
4	1	8	4	20

The data packet contains four bits that select bytes for partial-word stores and a parity bit in addition to the data. The request port thus has 37 bits of information. Two control bits are required, for a total of 39 bits.

The first response packet needs only the operation code, parity, PE number, and ORI; in this case the port width is determined by the width of the data itself so only 33 bits are required, plus 2 for control.

3.1.3. Network Performance

Overall system performance for very large numbers of processors critically depends on the message throughput achieved by the interconnection network. Since a PE may sometimes require the data from a previous request before it can make another request, message throughput is limited not only by the bandwidth of the network but by its latency, the time from the generation of a request until a response is received. As stated above, the available bandwidth grows linearly with the number of processors. The latency growth is more complicated and is discussed next.

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 Kruskal *et. al.* [86] (see also Percus and Percus [88]), 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 expected 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 has been obtained from simulations (see also Norton and Pfister [85] for a similar correction factor). Note the term quadratic in m. In addition to its effect on the average latency, Kruskal and Snir [83] show that even moderately large values of m have a large effect on the *variance* of the latency.

To determine the accuracy of these estimates with bounded buffer size and non-uniform traffic, we simulated networks composed of 2×2 switches under various assumptions of switch architecture, PE request generation, and memory behavior (see Dickey and Liu [88] for details). With infinite buffers the theoretical network bandwidth allows up to one packet per cycle per port. However, with finite buffers the *effective throughput* (the number of messages actually sent per cycle) will be less due to blocking. Various values of buffer size and *offered load* (the probability of that a PE will initiate a message given that it is neither blocked by a full queue nor already transmitting a message),[10] have been simulated to get estimates of effective throughput as well as latency.

These simulations have shown that, for two-packet messages and up to an offered load of about 0.3 (which for infinite buffers corresponds to 0.46 packets per cycle per port) buffers large enough to hold 6 messages give an effective throughput very close to the offered load. Round-trip latency is less than that predicted by the Kruskal *et al.* [86] formula given above. This decrease is due to smaller queuing delays on the return path, for which we have as yet no analytic explanation. Combining keeps latency low even with heavy hot-spot traffic in a 256-PE network. However, we do not know the effect of either non-uniform (but not hot-spot) traffic or ‘‘bursty’’ traffic.

In many practical cases, a PE may not issue another request until a previous request has been satisfied. The simulator can require a PE to wait for a previous load request to complete before making another. If one prefetch is allowed, a PE can have two load requests outstanding at the same time, and similarly for additional prefetching. Note that as the load increases, more prefetching greatly increases effective throughput (Figure 4).

3.2. Processing Element

This section starts with our broad goals for the PE and their impact on its overall design. Two important requirements for the PEs in a shared-memory parallel processor are maintaining a consistent view of data and reducing the cost of periodic clock interrupts; these are discussed in successive subsections. Finally, we present an overview of a PE design using the Am29000.

3.2.1. PE Design Goals

The most important goal is support for all the Ultracomputer architectural components. These components must be supported in a way that does not make their use *preferentially inefficient*; for example, supporting

[10] If we assume infinite buffering (so that a PE is never blocked), then on average each PE transmits $\frac{m}{m+(1-p)/p}$ packets per cycle, where m is the number of packets per message and p is the offered load.

Offered Load	Effective throughput by amount of prefetch				Latency by amount of prefetch			
	None	One	Two	∞	None	One	Two	∞
0.1	0.04	0.07	0.09	0.09	19.9	20.4	20.4	20.9
0.3	0.06	0.11	0.15	0.23	20.3	21.5	22.7	26.2
0.5	0.07	0.12	0.17	0.33	20.4	21.9	23.5	36.4
0.7	0.07	0.13	0.18	0.41	20.5	22.1	23.9	58.6

Figure 4. Effective throughput (messages per cycle) and latency (cycles) for varying prefetch and offered load: 256 PEs, 2×2 switches, 20-packet queues, two-packet messages, and memory cycle time of two network cycles. Request mix: 60% loads, 40% stores. Unlimited stores allowed.

fetch-and-add only via a system call will discourage its use. Specifically, the architectural features that must be supported are:

- fetch-and-ϕ operations available to application-level code
- multiple simultaneous operations outstanding in the network
- ‘‘fence’’ operations (section 3.2.3)
- application control of cachability and explicit cache invalidations

Our next most important goal is keeping the PE simple. Along with reducing the component count (and hence cost and size) of a PE, this will significantly lower the design cost; a very complex PE was a major reason the IBM RP3 project produced a slower-than-expected system two years late and was also the cause of a longer-than-expected debugging time for our previous prototype, Ultra II.

A PE with high performance encourages application developers to run their applications on our prototype, thus providing more ‘‘test data’’ for experimentation. However, unlike a commercial vendor, performance on an absolute measure is less important than support of the complete architecture and design simplicity because the primary purpose of our prototype is to provide a testbed for studies of the Ultracomputer architecture. Nevertheless, we expect the performance of a 256-PE machine to be quite respectable; future 4096-PE systems built with then-current microprocessors may well achieve among the highest sustained performance of any supercomputer at that time.

Finally, whenever not inconsistent with the above goals, we provide architectural flexibility to allow studies of the effect of architectural changes. For example, software TLB reload will allow studying scalable strategies for maintaining TLB consistency (see Teller *et al.* [88] for several strategies we are considering). In addition, implementing cache control using programmable logic allows subsequent research into more advanced cache features.

3.2.2. PE Design Issues

Given the desire for simplicity, it might at first appear that the best approach to a PE is to construct a network interface board for an off-the-shelf processor board (such as the Processor Module for the ES-Kit message-passing system, MCC [87], or one of the many commercially available boards that connect to a standard bus). Unfortunately, this is not true in our case. The fundamental reason is that there are a number of significant architectural differences between a PE designed for a uniprocessor (or message-passing) environment and one designed for our scalable, shared-memory environment (see Kenner *et al.* [89] for details); note that a PE implementing these features is not necessarily *more complicated* than one for a uniprocessor, just *different*. The two most relevant features are support for both multiple outstanding requests in the network and the fetch-and-ϕ primitives. Stores can always be pipelined into the network (subject to sequential consistency requirements; see below). However, producing multiple outstanding loads requires a processor board with a split-transaction external interface.

Since the fetch-and-ϕ operations are implemented by the memory module, the chosen processor board must have some way of indicating that a particular bus cycle is a fetch-and-ϕ operation, rather than a load or a store.[11] The difficulty of doing so, and, indeed whether it can be done at all, is strongly determined by the versatility of the off-board interface of particular processor boards.

The above concerns restrict significantly our choice of a processor board and make interfacing a ''standard'' processor board quite complex. In addition, these processor boards would likely contain unnecessary components such as a ROM monitor, local RAM, and I/O interfaces. Along with the inherent waste of having an extra level of interfaces, this would greatly increase the component count. The use of boards designed for other purposes would also produce mechanical compatibility problems with the network packaging. Although these problems would not be severe for the initial 16-PE prototype, using vastly different PE designs on the initial and subsequent prototypes would not permit the initial prototype to serve its purpose as an engineering testbed to validate the design for the large prototype; we wish to avoid the necessity of performing logic design in the construction of the 256-PE prototype.

The choice of a microprocessor is also subject to our prioritized goals stated above: support for all features of the Ultracomputer architecture, simplicity of resulting PE design, performance, and flexibility. These goals are, of course, often in conflict.

Raw performance, measured by some standard benchmark, is not our dominant performance concern. Instead, we focus on the number of cycles required to execute one (''standard'') instruction since our desire to use relatively simple technology will likely result in a system clock slower than the maximum clock frequency of the processor. The requirement that the number of cycles per instruction be small favors RISC-style microprocessors. We have mostly considered these microprocessors with the exception of the MC68030, a CISC microprocessor, which was also considered because of our previous use of the MC68010.

Normally, the hardest part of supporting the Ultracomputer architectural features is grafting fetch-and-ϕ onto the selected microprocessor. As microprocessor integration increases, more system components (core processor, floating-point unit, memory management unit, and cache) are being placed on a single chip. Although this integration clearly reduces the component count of the PE, it can make the implementation of fetch-and-ϕ quite complex. For example, implementing fetch-and-ϕ on the 68030 involves using the coprocessor interface in an unusual manner and depends on a detail of its microcode. We have found that processors designed for embedded applications tend to have more flexible interfaces and be more adaptable to our needs than those designed for a workstation environment.

There are also a number of non-technical issues. Since the implementation of some architectural components (most notably fetch-and-ϕ and multiple pending memory reads) requires using microprocessor features in uncommon ways, we require technical support from the microprocessor vendor. In addition, to facilitate software development, we need to obtain the operating system and compilers *with source code* for the selected microprocessor. In some cases, our choices were restricted by these non-technical issues. For example, at the time processor selection was being performed, we contacted MIPS Computer Systems about their processor family. After repeated inquiries, we were unable to obtain even a commitment to sell us chips, let alone make available operating system sources. Accordingly, we had to restrict our attention to parts from other vendors. (In any event, the MIPS microprocessor did not appear to be the best choice technically.)

In early 1988 we selected the AMD Am29000 processor family to be used on Ultra III based not only on its technical merits (in particular, the relative ease of a fetch-and-ϕ implementation, see section 3.2.5) but also on the fact that we received exceptional cooperation from AMD, including their making key

[11] An alternative implementation provides three registers on the interface board, two of which hold the address and operand of the fetch-and-ϕ, and one which acts as a ''trigger'' which is read to obtain the fetch-and-ϕ result. Since the network requires that a physical address be transmitted and the interface board would not be able to perform an address translation operation, the processor would have to transmit the physical address as data. This address may change due to paging and swapping. Hence fetch-and-ϕ cannot be done in user mode and requires a system call to do the address translation. As discussed above, this would produce an unacceptable disincentive for using the fetch-and-ϕ operations.

technical personnel available to us along with test equipment, software, and early chip samples.[12] We have attached a letter from AMD showing their commitment to provide all of the Am29000-family components required for the 16 processor prototype.

The Am29000 is a RISC microprocessor that achieves 17 MIPS with a 25 MHz clock. Included in the family is the Am29027, a floating-point coprocessor that performs at 3.1 (single-precision) LINPACK MFLOPS. (The Am29000 has been shipping for over a year and the Am29027 has been shipping since late 1988.) Since a new Am29000 family part will be available within a year that contains an embedded floating-point unit and is hardware and software compatible with the Am29000, we do not anticipate using the Am29027, but will do so if the new processor is delayed. Section 3.2.5 highlights the advantages of the Am29000.

3.2.3. Maintaining Data Consistency

If two PEs have the same 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 (Smith [82]). Although we and others are investigating the potential of using directory-based methods to aid in maintaining cache consistency on systems containing interconnection networks, much work remains to be done in this area. This work is outside the scope of this proposal. However, should progress be made in the near future, we will consider using these methods on our 256-PE prototype. Barring such advances, we prohibit the caching of shared read-write data unless the software specifically permits it and either takes responsibility for maintaining cache consistency or certifies that it is unnecessary.

Even in the absence of caches, there is an additional potential for data inconsistency because the network does not preserve FIFO ordering; see Collier [81], Lamport [79], Lynch and Fisher [81], and Shasha and Snir [88] 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;	MyY ← Y;
Y ← 1;	MyX ← X;

Although we cannot predict the values of MyX and MyY, the case that MyX is 0 and MyY is 1 is incorrect. To prevent such behavior, the *fence* operation must be supported by the PE. This operation is also used by the RP3 (Brantley *et al.* [85]) and forces a PE to wait until all requests issued to the network have been acknowledged. Fences are required in both tasks **A** and **B** above; Shasha and Snir [88] discuss the general case in which they are required.

Most fence operations need wait only for the acknowledgement of requests involving noncachable data since cachable data is not considered shared read-write for the reasons discussed above. However, when a context switch occurs, all modified data must be written to memory before another PE can execute the process. As in the RP3, two fence operations, NFENCE and AFENCE, are defined and implemented in the PE hardware; 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.

[12] Subsequent to that date, the Intel i860 was announced. We considered abandoning our investment in the 29000 and switching to the i860. However, our analysis showed that, although the processor has the potential of being significantly faster than the 29000 given sufficient compiler technology, there were a number of disadvantages of the i860. One was the complexity of interfacing to the processor, especially the difficulty of implementing fetch-and-ϕ. Another was that we would probably not be able to use its on-chip data cache because its structure complicated software approaches to maintaining cache consistency. Finally, the i860 was less flexible that the 29000; e.g., software TLB reload was not provided. Therefore, we decided that, on balance, it would not be justified to abandon our investment in the 29000.

3.2.4. Reducing the Cost of Periodic Interrupts

Periodic clock interrupts, conventional in uniprocessor systems, have a higher 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 reduced by combining, tend to saturate the network.[13]

Clock interrupts are used to maintain the time-of-day and provide interval timing (such as CPU quantum expiration). In the absence of explicit interprocessor interrupts, multiprocessor operating systems also use clock interrupts to check periodically for events such as the addition of a process to the ready queue or the delivery of a signal to a running process. The requirements for such periodic polling would tend to increase the desired clock interrupt rate, and hence their overhead. In order to eliminate the need for these periodic interrupts, we discuss alternative methods of providing the same functionality.

Time maintenance and interval timing can be performed by the Am29000 itself. The Am29000 contains a register that is incremented each clock cycle. 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* (described in Edler *et al.* [85]) communicates with the PE running a specific process, to deliver a **signal** to that process. Each time a PE selects a process to run, the PE number is placed in a known location, called a *reflector*, associated with that process. When the process is not running, the reflector is set to a value indicating that it is disabled. To **signal** a process that is free to migrate among PEs, a **reflect** operation is directed at the process's reflector. This operation is received by the MM containing the reflector, which reads its value and returns this information to the originating PE. If the process is running, the MM sends an additional message to interrupt the PE running the process and cause it to deliver the signal. If the requester receives a response indicating that the reflector is disabled, it handles delivery itself. Additional higher-level synchronization is required for the case where the process is picked up by another PE before the requester receives its response; see Edler, *et al.* [85] for details.

The second mechanism provides a number of bussed interrupt lines. Each line is received by all PEs and can be driven by any of them or by an external processor. No guarantee is given either that an interrupt that is requested will occur or that a spurious interrupt will not occur. This makes the interrupt advisory and hence a perfectly reliable medium for transmission is not required. Thus we can use, for example, TTL ''open collector'' techniques to implement a bus that any PE can pull to ground to cause an interrupt. The fact that this communication is not required to be reliable and that multiple interrupt requests can (and, in fact, *should*) be converted into a single interrupt allows the use of an inherently unscalable bus in this situation.

Each PE has a mask bit for each line to enable selectively which lines may interrupt it. 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 interval timer 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 an actual hardware interrupt.

3.2.5. PE Design Overview

The design of a PE using the Am29000 is discussed in this section and is shown in Figure 5.

To implement fetch-and-ϕ on the Am29000, we take advantage of the fact that an additional bit, the *lock* bit, is present in each **load** and **store** instruction. This bit can be set by the user and has no internal

[13] The concern is instruction fetches for the clock interrupt code itself, in the case where it is not in a cache. Unlike the situation with a single hot-spot, for which combining is effective, here we have a number of locations, each of which is referenced by all PEs. If they execute in lockstep, combining eliminates the resulting network contention. However, if they drift away from lockstep execution, the rate of combining decreases. The subtle feedback effects that result in this case are poorly understood; it is not obvious whether they cause the PEs' execution to tend more towards lockstep or to diverge. We suspect the latter, but are not aware of any work in this area.

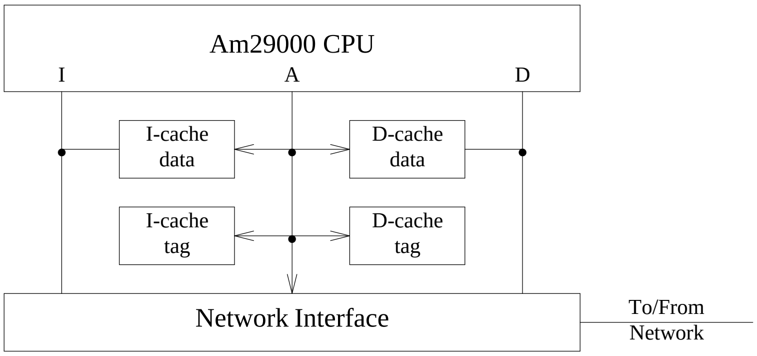

Figure 5: PE Board Structure

effect, but is presented externally when the request is issued. A fetch-and-ϕ is performed by first setting the operation type and operand with an I/O-space **store**. A **load** instruction with the *lock* bit set is then issued. The target address is translated to a physical address by the normal operation of the load instruction.

A PE using the Am29000 can have many requests simultaneously outstanding in the network. The Am29000 has three 32-bit buses: address, instruction, and data. Addresses need not be presented during the entire transaction, enabling instruction and data references to proceed simultaneously. It also supports a ‘‘burst mode’’ where a single address is presented and subsequent accesses reference consecutive locations. 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, up to three simultaneous load requests can be issued by the Am29000: one primary request on each port (instruction and data) and one pipelined request on one of the ports. For burst requests a network interface can prefetch additional words in advance of their being requested. Finally, stores can often be acknowledged once they are transmitted to the network. Therefore, a substantial number of outstanding requests from each PE can be in the network simultaneously.

The Am29000 has other features that minimize the effects of long memory latency for both instruction and data fetches. It prefetches instructions into a four-word buffer, essentially pipelining instruction fetches. 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 Am29000 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 second load is placed on the bus as a pipelined request.

Analytical models of the extent to which the 29000’s architecture masks latency are not available, but utilizations in the range of 50-75% seem likely. These utilizations are based on extrapolations from modeling done with the simpler but less latency-tolerant MC68030 and translate into aggregate sustained performance for a 256-PE Ultra III in the range of 2000-3200 MIPS at a 25 MHz system clock speed.

The PE consists of the Am29000 processor, instruction and data caches, a network interface, and control logic. For simplicity, the caches are direct-mapped and write-through. Chips such as the IDT7174 that combine static RAM with a comparator are used for the tag storage and conventional SRAMs are used for data storage. Each cache contains 8K lines, the size of the IDT7174.

The instruction and data caches have different goals. On the instruction side, it is important to provide 32 bits of instructions each cycle whenever the accesses are sequential (made using the Am29000’s burst mode). By organizing the instruction cache as a 8K$\times$64 array, 64 bits can be read from the SRAMs every other cycle. This makes their cycle time less critical while allowing 32 bits to be presented to the processor each cycle.

For data caches, burst mode accesses are relatively rare (mainly register stack spills and fills) and 32-bits per cycle bandwidth is not required. Also, straightforward implementation of software consistency strategies requires each cache line to consist of only one word; this is also the simplest design as it does not have to deal with partially-valid lines (which cannot occur with the read-only instruction cache). Therefore, the data cache is 8K×32 bits and presents a word one cycle after it has been requested.

Each tag store will contain 16 bits (validity bit plus 15 address bits), so two IDT7174s are required for each cache, for a total of four. Using 8K×8 SRAMs (e.g., IDT7164), an additional 12 chips are required (eight for the instruction cache and four for the data cache). Therefore a total of 16 chips are used to provide both a 64KB instruction cache and a 32KB data cache.

Each ‘‘side’’, instruction and data, has its own state machine. These state machines are responsible for determining hit or miss, making network requests, and handling responses from the network. Arbitration is performed at the outgoing network port, which also assigns an ORI value to each outgoing request; responses are directed to the appropriate side by consulting the ORI. The data side is fairly straightforward; the instruction side prefetches data on the network until the maximum number of outstanding requests is reached or the processor executes a branch instruction.

The control logic, including both state machines, and some of the data paths is implemented using XILINX programmable gate arrays; we anticipate one or two such chips per PE. Using programmable logic enables a large variety of experiments to be performed. For example, we can regulate the number of outstanding requests and the degree of prefetch permitted. We can also experiment with the effects of different interleaves or other hashing functions. Many other interesting experiments will no doubt also be possible. Finally, the XILINX chips can be programmed to count various events for instrumentation purposes (though the amount of information obtained by this manner is limited).

3.3. MM Design

The function of the MM is straightforward: it receives requests from the network and responds to them.[14] Overall performance is influenced by the MM’s cycle time (the reciprocal of its service rate). Specifically, the MMs must be able to provide an aggregate service rate greater than the aggregate request rate of the PEs (or more specifically the caches). In contrast, any latency added by the MM will be small compared to network latency. That is, it will be the memory cycle time and not its access time that will most strongly determine performance.

Memory cycle time is the reciprocal of the clock rate times the number of cycles needed to process a request. The large prototype will have a clock rate of 20-30 MHz, i.e., packets arrive every 33-50 ns. Since the network requires all requests to contain an even number of packets, messages can arrive every 66-100 ns. For simplicity we specify that the MM cycle time be a multiple of this message interarrival time. The major limitation on the MM cycle time is that of the memory chips.[15] The use of relatively inexpensive DRAMs provides MM cycle times in the range of 150-200 ns and thus we set the MM cycle time at twice the interarrival time or four network clock cycles, which simulations have shown to be adequate.

The instruction cache issues load double requests, which generate four packet responses. Since the MM is organized as a 1Mb×32 (plus parity) array, a 64-bit load requires two accesses. Using ‘‘nibble-mode’’ to provide the second 32-bit word permits the DRAMs to complete these loads in four cycles. Fetch-and-ϕ operations requiring use of the ALU cannot be completed in four cycles; these require six cycles.

To permit loads to be combined with fetch-and-adds, the PEs transmit 32-bit loads as fetch-and-adds with an increment of zero; this does not increase network traffic since the minimum message length is two

[14]The MM also has a second port that connects to a back-end I/O processor. Because of the synchronous nature of the entire system, arbitration between the ports is not in the critical path and can be done in parallel with servicing requests.

[15]The effective bandwidth of an MM can be increased by constructing multiple banks of RAMs in each MM but doing so requires additional logic to route requests to the appropriate bank. A more efficient way of achieving the same effect is adding additional stages to the network and having two or four times as many MMs as PEs.

packets. To perform 32-bit loads in four cycles, the MM suppresses the write operation for fetch-and-adds with zero increments. The analogous test is also performed for fetch-and-or and fetch-and-andnot.[16] Similarly, a 32-bit store is transmitted as a fetch-and-store and is also performed in four cycles.

Figure 6 summarizes the requests processed by the MM. It contains the size, in packets, of the request and response messages and the number of cycles required by the MM for each operation. It also indicates which operations are combined by the network.

3.4. Packaging

This section discusses the packaging of a 256-PE Ultra III with a combining network. The packaging of both the switch nodes and the complete system is discussed.

3.4.1. Switch Packaging

Each forward-path port requires 39 bits (37 for data and 2 for control); each return-path port 35. Thus, a single-chip 2×2 switch would require over 300 signal pins, a large number using today's technology. With two chips per switching node, one for each direction, the maximum pin count is reduced. The forward path chip requires about 230 signal pins, the return path 210. These are within the capability of a single chip using modern packaging.

The final switch packaging will depend on the technology available to us, possibly from our industrial partners. As discussed below, we expect to construct a 4×4 switch on a single board. This board would contain four 2×2 switches for a total of eight chips.

3.4.2. System Packaging

A simple modular packaging scheme for Ultracomputers has been developed using Cinch's ''Button'' technology (Smolley [85]) and MCC's ''ES-kit'' packaging system (MCC [87]). This technology facilitates the interconnection of stacked modules and provides separate sets of connections on the top and bottom faces of each module. With appropriate network partitioning, a simple set of Ultra modules are sufficient to assemble Ultracomputers of different sizes. The attached report, Bianchini [89], ''Ultracomputer Packaging and Prototypes'', discusses this packaging in detail; here we summarize the key points, referring to the diagrams in the attached report. Manufacturer's specifications and descriptions of the technology used (called CIN•APSE™) are also attached.

Operation	Req. Size	Resp. Size	Service Time	Comb-ined?
Load Double	2	4	4	√
Fetch-and-Add	2	2	6(4)	√
Fetch-and-Or	2	2	6(4)	√
Fetch-and-AndNot	2	2	6(4)	√
Fetch-and-Store	2	2	4	√
Fetch-and-Store-if-Zero	2	2	4	
Fetch-and-Store-if-≥0	2	2	4	
Partial-Word-Store	2	2	4	
Reflect	2	4	4	

Figure 6. Requests processed by the MM

[16] The fetch-and-andnot operation performs a logical **and** of memory with the ones complement of the transmitted data (a selective bit clear). Fetch-and-andnot is used instead of fetch-and-and to simplify the design of the switch: both fetch-and-or and fetch-and-andnot are combined by performing a logical **or** of their arguments.

A Button contact is fabricated from strands of 2 mil diameter silver-copper wire randomly pressed to form a gold-plated cylinder 40 mils in diameter. One Button is attached to each end of a conducting rod about an inch long and connects to 50-mil diameter gold-plated ‘‘lands’’ on adjacent PC-boards. Each Button makes multiple contacts at the interface surface and provides multiple conducting paths through the structure. Contacts are under high pressure due to the spring created by the wadded wire and the small size of the contact area. Control of the spring constant and contact pressure can be achieved by controlling the length of wire used to make the Button. Only pressure is required to complete the circuit (approximately two ounces per Button). The assembly has a contact resistance of less than a third of a milliohm and is electrically invisible at frequencies up to that of microwaves. It can be used to distribute either power or signals.

A controlled-impedance connector developed jointly by Cinch and MCC for MCC’s ES-Kit project provides four rows of Buttons on 100 mil centers with a 50 mil offset between rows. This connector consists of 106 rods as described above and has a ‘‘footprint’’ of 3.5″×0.5″, including alignment pins and bolt holes. With these connectors, standard PC-board fabrication technology is used to produce an appropriate pattern of 50-mil lands, and holes are drilled for alignment pins and compression bolts.

These patterns can be placed anywhere on either of the board surfaces. If both surfaces are used, plated-through holes can be used to bus some or all of the signals from one side of the board to the other. On the standard ES-Kit 5.5″×7″ board, for example, each 7″ side of the board contains two connectors on each surface, for a total of 848 connections per board (424 if all signals are bussed between the surfaces of the board). The number of connectors that can be placed on a board is limited only by the space available on the board; this allows for extremely general interconnection schemes. Not all boards in a system need have connectors at the same places; for example, a dedicated set of connectors can be used to connect two adjacent boards.

There are many ways to construct Ultracomputers using this technology. They can be evaluated by comparing the total number of boards required, the total number of board *types* required, the total number of Buttons required, the size of the largest board, or similar criteria.

An omega network of N ports has the property that, for any power of two, $i \le \sqrt{N}$, a block containing an $i \times i$ sub-network and either i PEs or i MMs can be constructed where all the switch connections are internal to the block and i ports exit from the block. In the case of a 256-PE Ultra III, we can construct basic blocks of four PEs or MMs that include a 4×4 sub-network. We package such a block on 5 boards, one for each PE (or MM) and one for the subnetwork.

Three basic module types are necessary to form a four-processor Ultracomputer: a processor module (PE), a memory module (MM), and a four-input, four-output switch module (SW). Using conservative packaging, each Ultracomputer module fits on a 10″×7″ board (Figure A1 in the attached Bianchini report), which is about twice the size of the standard ES-Kit board. As on the ES-Kit boards, each 7″ side contains a pair of connector patterns on each board surface. Since each network port (PE-to-MM plus MM-to-PE paths) requires 74 wires and each connector provides 106 contacts, one connector is used for each port. Each board therefore has four ports per surface. We construct a board containing four 2×2 switches connected to form a 4×4 sub-network with its PE-side ports (i.e., the bidirectional ports receiving memory requests and transmitting memory responses) on one surface and its MM-side ports on the other.

Both the PE and MM modules wire their external ports both to one corner of the top face and to a corner of the bottom face rotated by 90 degrees. The remaining three top corner connections are separately wired to a 90 degree rotated bottom corner (Figure 3.1 in the attached report). The ports to the network from four stacked PEs or MMs are thus routed through the intermediate boards to both the top and bottom faces of the stack. The SW module has its PE-side ports wired to each of the four top corners and its MM-side ports to the corresponding four bottom corners. A stack of nine modules (4 PEs, 1 SW and 4 MMs) comprises a four-PE Ultracomputer, a useful assembly for diagnosis of PE/MM operation. It is shown in Figure 3.4 of the attached report.

A 256-processor system (Figure 4.10 in the attached report), constructed using this interconnection method, has 16 boards in a stack. Its dimensions are 80″×56″×16″. The largest board in this system is

$80'' \times 7''$, which is impractically large. As discussed in Section 5 of the attached report, this board may be replaced with three $40'' \times 7''$ boards, each of which can, in turn, be replaced by three $20'' \times 7''$ boards. Since these boards contain no active components, they can be tightly packed.

For larger systems, we allow the use of technology that at this time is considered experimental or just coming into production use. Specifically, we assume a denser connector providing 8 ports per board, tape-automated bonding (TAB) allowing a pitch of 15 mils between board contacts, and extensive use of surface-mount technology. This allows an 8×8 switch, a PE, or an MM to be packaged on a single $5.5'' \times 7''$ ES-Kit board. The basic vertical unit is then a stack of 8 PEs, an 8×8 switch, and 8 MMs. A 4096-PE system requires 512 such stacks, arranged in a 32×16 array. Each stack contains four stages of 8×8 switches and a number of interconnection boards. The system occupies a volume of $15' \times 9' \times 3'$. Techniques similar to those above are used to eliminate the need for $15'$ boards. Use of more aggressive technologies will produce even smaller systems.

4. References

George B. Adams, III and Howard Jay Siegel, ‘‘The Extra Stage Cube: A Fault-tolerant Interconnection Network for Supersystems’’, *IEEE Trans. Comp.* **C-31** No. 5, 1982, pp. 443-454.

Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante, ‘‘An Overview of the PTRAN Analysis System’’, *J. Parallel and Distributed Comp.*, 1988, pp. 617-640.

J. Randy Allen, David Callahan, and Ken Kennedy, ‘‘Automatic Decomposition of Scientific Programs for Parallel Execution’’, *Conf. Record, Tenth Ann. ACM Symp. on Principles of Prog. Lang.*, 1983.

Prithviraj Banerjee and Abhijeet Dugar, ‘‘The Design, Analysis, and Simulation of a Fault-Tolerant Interconnection Network Supporting the Fetch-and-add Primitive’’, *IEEE Trans. Comp.* **38** No. 1, Jan. 1989, pp. 30-46.

Wayne Berke, ‘‘Pdb–An Ultracomputer Debugger for Parallel Programs–User’s’’ Guide", Ultracomputer Documentation Note #3, Courant Institute, NYU, 1986.

Wayne Berke, ‘‘ParFOR—A Structured Environment for Parallel FORTRAN’’, Ultracomputer Note #137, Courant Institute, NYU, 1988.

Ronald Bianchini, ‘‘Ultracomputer Packaging and Prototypes’’, Ultracomputer Note #152, Courant Institute, NYU, 1989.

Marvin Bishop, ‘‘The Ultracomputer as a Vehicle for Polymer Simulations’’, NYU Ultracomputer Note #63, 1983.

William C. Brantly, Kevin P. McAuliffe, and Jodi Weiss, ‘‘RP3 Processor-Memory Element’’, *Proc. ICPP*, 1985, pp. 782-789.

Jerome Chiabaut, Vladimir Fleyshgakker, and Anne Greenbaum, ‘‘Porting Scientific Applications to the NYU Ultracomputer’’, NYU Ultracomputer Note #144, 1988.

William W. Collier, ‘‘Principles of Architecture for Systems of Parallel Processes’’, IBM Technical Report TR00.3100, March 1981.

W. Crowther, J. Goodhue, R. Gurwitz, Randall Rettberg, and R. Thomas, ‘‘The Butterfly (TM) Parallel Processor’’, *IEEE Comp. Arch. Tech. Committee Newsletter*, Sept.-Dec. 1985a, pp. 18-45. This issue contains several other articles on the Butterfly.

W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, ‘‘Performance Measurements on a 128-Node Butterfly Parallel Processor’’, *Proc. ICPP* 1985b, pp. 531-540.

Frederica Darema, Dave George, Alan Norton, and Gregory F. Pfister, ‘‘A Single-Program-Multiple-Data Computational Model for EPEX FORTRAN’’, *Parallel Computing* **7** No. 1, April, 1988, pp. 11-24.

Frederica Darema-Rogers, S. Kirkpatrick, and Alan Norton, ‘‘Simulated Annealing on Shared Memory Parallel Systems’’, *IBM J. of R&D*, 1987a.

Frederica Darema-Rogers and Gregory F. Pfister, ‘‘Multipurpose Parallelism in VLSI Computer Aided Design: A Case Study’’, Tech. Rept., IBM T.J. Watson Research Center, 1986.

Frederica Darema-Rogers, Gregory F. Pfister, and K. So, ‘‘Memory Access Patterns of Parallel Scientific Programs’’, *Proc. Sigmetrics Conf.*, 1987b.

Susan Dickey, Richard Kenner, Marc Snir, and Jon Solworth, ‘‘A VLSI Combining Network for the NYU Ultracomputer’’, *Proc. Intern. Conf. on Comp. Design*, 1985, pp. 110-113.

Susan Dickey and Yue-Sheng Liu, ‘‘Simulation and Analysis of Different Switch Architectures for Interconnection Networks in MIMD Shared Memory Machines’’, Ultracomputer Note #141, Courant Institute, NYU, 1988.

Isaac A. Dimitrovsky, ‘‘ZLISP—A Portable Lisp System’’, Ph.D. Thesis, Courant Institute, NYU, 1988.

Anne Dinning and Bud Mishra, ‘‘Fully Parallel Algorithms for Implementing Path Expressions’’, Ultracomputer Note #154, Courant Institute, NYU, 1989.

Jan Edler, ‘‘Readers/Readers Synchronization’’, Ultracomputer System Software Note #46, Courant Institute, NYU, 1984.

Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, Paticia J. Teller, and James Wilson, ‘‘Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach’’, *Proc. 12th Annual Intern. Symp. on Comp. Arch.*, 1985, pp. 126-135.

Jan Edler, Jim Lipkis, and Edith Schonberg, ‘‘Process Management for Highly Parallel UNIX Systems’’, *Proc. Usenix Workshop on UNIX and Supercomputers*, Pittsburgh, 1988a, pp. 1-17.

Jan Edler, Jim Lipkis, and Edith Schonberg, ‘‘Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors’’, *Proc. Usenix Workshop on UNIX and Supercomputers*, Pittsburgh, 1988b, pp. 151-168.

Tse-yun Feng and Chuan-lin Wu, ‘‘Fault Diagnosis for a Class of Multistage Interconnection Networks’’, *IEEE Trans. Comp.* **C-30** No. 10, 1981, pp. 743-758.

Susan Flynn, Edith Schonberg, Edmond Schonberg, ‘‘The Efficient Termination of Ada Tasks in a Distributed Environment’’, *Proc. HICS*, Jan. 1987.

Susan Flynn, ‘‘SMARTS—Shared-memory Multiprocessor Ada Run Time Supervisor’’, Ph.D. Dissertation, Courant Institute, NYU, Dec. 1988.

Eric Freudenthal and Allan Gottlieb, ‘‘Process Coordination with Fetch-and-Increment’’, in preparation, 1989.

Eric Freudenthal and Olivier Peze, ‘‘Efficient Synchronization Algorithms Using Fetch-and-Add on Multiple Bitfield Integers’’, Ultracomputer Note #148, Courant Institute, NYU, 1988.

Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh, ‘‘Cedar’’ in *Supercomputers: Design and Applications*, Kai Hwang, ed. 1984.

Lance A. Glasser and Charles A. Zukowski, ‘‘Continuous Models for Communication Density Constraints on Multiprocessor Performance’’, *IEEE Trans. Comp.* **C-37** No. 6, 1988, pp. 652-656.

G.R. Goke and G.J. Lipovski, ‘‘Banyan Networks for Partitioning Multiprocessor Systems’’, *Proc. 1st Annual Intern. Symp. on Comp. Arch.*, 1973, pp. 21-28.

James R. Goodman, Mary K. Vernon, Philip J. Woest, ‘‘Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessing’’, *Proc. ASPLOS III*, Boston, April, 1989, pp. 64-73.

Allan Gottlieb, ‘‘PLUS - A PL/I Based Ultracomputer Simulator, I and II’’, Ultracomputer Notes #10 and #14, Courant Institute, NYU, 1980a.

Allan Gottlieb, ‘‘Washcloth—The Logical Successor to Soapsuds’’, Ultracomputer Note #12, Courant Institute, NYU, 1980b.

Allan Gottlieb, ‘‘An Overview of the NYU Ultracomputer Project’’, *Experimental Parallel Computing Architectures*, Jack J. Dongarra, ed., North Holland, 1987, pp. 25-95. An updated version is available as Ultracomputer Note #100, Courant Institute, NYU, 1987.

Allan Gottlieb and Clyde P. Kruskal, ‘‘MULT - A Multitasking Ultracomputer Language with Timing, I & II’’, Ultracomputer Note #15, Courant Institute, NYU, 1980.

Allan Gottlieb and Clyde P. Kruskal, ‘‘Complexity Results for Permuting Data on Parallel Processors’’, *JACM* **31** No. 2, April 1984, pp. 193-209.

Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Lawrence Rudolph, and Marc Snir, ‘‘The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer’’, *IEEE Trans. Comp.*, pp. 175-189, February 1983a.

Allan Gottlieb, Larry Rudolph, and Boris Lubachevsky, ‘‘Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors’’, *ACM TOPLAS* **5** No. 2, 1983b, pp. 164-189.

Anne Greenbaum, ‘‘Solving Sparse Triangular Linear Systems Using FORTRAN with Parallel Extensions on the NYU Ultracomputer Prototype’’, Ultracomputer Note #99, Courant Institute, NYU, 1986.

Anne Greenbaum, Congming Li, and Han Zheng Chao, ‘‘Comparison of Linear System Solvers Applied to Diffusion-Type Finite Element Equations’’, Ultracomputer Note #126, Courant Institute, NYU, 1987, to appear in *Num. Math.*

Malvin Kalos, ‘‘Scientific Calculations on the Ultracomputer’’, Ultracomputer Note #30, Courant Institute, NYU, 1981.

Malvin H. Kalos, Gabi Leshem, Boris D. Lubachevsky, ‘‘Molecular Simulation of Equilibrium Properties. Parallel Implementation’’ NYU Ultracomputer Note #27, 1982.

Malvin H. Kalos and Y-Q. Zhong, ‘‘Monte Carlo Transport Calculation on an Ultracomputer’’, NYU Ultracomputer Note #46, 1983.

Richard Kenner, Susan Dickey, and Patricia J. Teller, ‘‘The Design of Processing Elements on a Multiprocessor System with a High-Bandwidth, High-Latency Interconnection Network’’, *Proc. HICSS*, 1989, Vol. I, pp. 319-328.

David Korn, ‘‘Converting Scientific Codes to Run Under the Washcloth Simulator’’, Ultracomputer Note #23, Courant Institute, NYU, 1981a.

David Korn, ‘‘Timing Analysis for Scientific Codes Run under Washcloth Simulation’’, Ultracomputer Note #24, Courant Institute, NYU, 1981b.

David Korn, ‘‘Timing Simulations for Elliptic PDE’s Run Under Washcloth’’, Ultracomputer Note #31, Courant Institute, NYU, 1981c.

David Korn and Norman Rushfield, ‘‘Washcloth Simulation of Three-Dimensional Weather Forecasting Codes’’, Ultracomputer Note #55, Courant Institute, NYU, 1983.

Clyde P. Kruskal and Marc Snir, ‘‘The Performance of Multistage Interconnection Networks for Multiprocessors’’ , *IEEE Trans. Computers* **C-32**, pp. 1091-1098, 1983.

Clyde Kruskal, Marc Snir, and Alan Weiss, ‘‘The Distribution of Waiting Time in Clocked Multistage Interconnection Networks’’, *Proc. ICPP*, 1986, pp. 12-19. A revised version appears in *IEEE Trans. Comp.* **37** No. 11, Nov. 88, pp. 1337-1352.

David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe, ‘‘Dependence Graphs and Compiler Optimizations’’, *Conf. Record, Eighth Ann. ACM Symp. on Principles of Prog. Lang.*, 1981.

Leslie Lamport, ‘‘How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs’’, *IEEE Trans. Comp.* **C-28** No. 9, Sept. 1979, pp. 690-691.

Duncan H. Lawrie, ‘‘Access and Alignment of Data in an Array Processor’’, *IEEE Trans. on Comp.*, **C-24** No. 12, Dec. 1975, pp. 1145-1155.

Cyungho Lee, Clyde P. Kruskal, and David J. Kuck, ‘‘The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of ‘Hot Spots’’’, *Proc. ICPP*, 1986, pp. 35-41.

J. Edward Lilienkamp, Duncan H. Lawrie, and Pen-Chung Yew, ‘‘A Fault Tolerant Interconnection Network Using Error Correcting Codes’’, *Proc. ICPP*, 1982, pp. 123-125.

G.J. Lipovski and Paul Vaughan, ‘‘A Fetch-And-Op Implementation for Parallel Computers’’, *Proc. 15th Annual Intern. Symp. on Comp. Arch.*, 1988, pp. 384-392.

Yue-Sheng Liu, ‘‘Delta Network Performance and ‘Hot Spot’ Traffic’’, Ultracomputer Note #132, Courant Institute, NYU, Jan. 1988.

Yue-Sheng Liu and Susan Dickey, ‘‘Simulation and Analysis of Enchanced Switch Architectures for Interconnection Networks in Massively Parallel Shared Memory Machines’’, *Proc. 2nd Symposium on the Frontiers of Massively Parallel Computation*, Oct. 1988, to appear.

Boris D. Lubachevsky, ‘‘Parallelizing an Algorithm of Charles S. Peskin for describing Incompressible Flow for a Fluid Coupled to an Elastic Membrane’’, NYU Ultracomputer Note #42, 1982.

Nancy A. Lynch and M. J. Fisher, ‘‘Semantics of Concurrent Computation’’, *Theor. Comp. Sci.* **13** No. 1, Jan. 1981, pp. 17-43.

Kevin McAuliffe, ‘‘Analysis of Cache Memories in Highly Parallel Systems’’, Ph.D. thesis, Courant Institute, NYU, 1986.

MCC, ‘‘ES-Kit Project Overview’’, Technical Report Number: ACA-ESP-391-87, December 21, 1987.

V. Alan Norton and Gregory F. Pfister, ‘‘A Methodology for Predicting Multiprocessor Performance’’, *Proc. ICPP*, 1985, pp. 772-781.

Ora E. Percus and Jerome K. Percus, ‘‘Elementary Properties of Clock-regulated Queues’’, Ultracomputer Note #138, Courant Institute, NYU, May 1988. Submitted to *SIAM J. on Appl. Math*.

Ora E. Percus and Jerome K. Percus, ‘‘Models for Queue Length in Clocked Queuing Networks’’, Ultracomputer Note #153, Courant Institute, NYU, Jan. 1989. Submitted to *Comm. Pure and Appl. Math*, 1989.

Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfielder, Kevin P. McAuliffe, Evelin S. Melton, V. Alan Norton, and Jodi Weiss, ‘‘The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture’’, *Proc. ICPP*, Aug. 1985, pp. 764-771.

Gregory F. Pfister and V. Alan Norton, ‘‘‘Hot Spot’ Contention and Combining in Multistage Interconnection Networks’’, *IEEE Transactions on Computers*, pp. 943-948, October 1985.

Randall Rettberg, private communication, 1987.

Norman Rushfield, ‘‘Atmospheric Computations on Highly Parallel MIMD Computers’’, Ultracomputer Note #22, Courant Institute, NYU, 1981.

Jacob T. Schwartz, ‘‘Ultracomputers’’, *ACM TOPLAS*, **2**, No. 4, Oct. 1980, pp. 484-521.

Edith Schonberg, ‘‘On-The-Fly Detection of Access Anomalies’’, *Sigplan ’89 Conf. on Prog. Lang. Design and Implementation*, to appear, 1989.

Dennis Shasha and Marc Snir, ‘‘Efficient and Correct Execution of Parallel Programs That Share Memory’’, *ACM TOPLAS*, **10**, No. 2, April 1988, pp. 282-312.

Alan Jay Smith, ‘‘Cache Memories’’, *ACM Computing Surveys*, **14**, pp. 473-530, September 1982.

Burton J. Smith, ‘‘Architecture and Applications of the HEP Multiprocessor Computer System’’, *Proc. SPIE—Intern. Soc. for Optical Eng.* **298** (Real-Time Signal Processing IV), San Diego, 1981. Reprinted in *Supercomputers: Design and Applications*, Kai Hwang, ed. 1984.

R. Smolley, ‘‘Button Board: A New Technology Interconnect for 2 and 3 Dimensional Packaging’’, TRW Technical Report, presented at Int. Soc. for Hybrid Microelectronics, Nov. 1985.

Marc Snir, ‘‘‘Netsim’ Network Simulator for the Ultracomputer’’, NYU Ultracomputer Note #28, 1981.

Marc Snir, ‘‘On Parallel Search’’, *Principles of Distributed Computing*, Aug. 1982.

Patricia J. Teller, ‘‘TLB Consistency and Demand Paging on a Highly-Parallel Shared-Memory Multiprocessor’’, Ph.D. Dissertation, NYU, 1989, in preparation.

Patricia J. Teller, Richard Kenner, and Marc Snir, ‘‘TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors’’, *Proc. HICSS*, 1988, Vol. I, pp. 184-193.

James Wilson, ‘‘Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add’’, Doctoral Dissertation, NYU, 1988.

David Wood, ‘‘Parallel Queues and Pools, an Evaluation’’, Ultracomputer Note #150, Courant Institute, NYU, 1989.

Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie, ‘‘Distributing Hot-spot Addressing in Large-scale Multiprocessors’’, *Proc. ICPP*, 1986, pp. 51-58.

Supporting Letters from Industrial Partners

The following pages contain copies of letters received from our industrial partners, AMD and MCC.

- A letter of support from AMD’s director of the Am29000 product family.
- A FAX of a letter from BYTE Magazine to AMD correcting an error in the May 1989 issue which gave the impression that AMD had discontinued the Am29000 family.
- A letter of support from MCC’s director of government programs.

Table of Contents