The Effects of Latency and Occupancy in Distributed Shared Memory Multiprocessors

Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy

(submitted to JPDC, also Stanford University Technical Report CSL-TR-95-660)

Abstract

Many designers of distributed shared memory (DSM) multiprocessors are proposing the use of commodity parts, not only in the processor and memory subsystem but also in the communication architecture. While the desire to use commodity parts in the communication architecture offers potential advantages in cost and design time, the impact on the performance of applications is unclear. In this paper we study this performance impact through detailed simulation and analytical modeling, using a range of important applications and computational kernels.

We characterize the communication architectures of DSM machines by four parameters, similar to those in the *logP* model. The *l* (latency) and *o* (controller occupancy, or controller bandwidth) parameters are the keys to performance in these machines, with the *g* (gap or node-to-network bandwidth) parameter not being a bottleneck in recent and upcoming machines. Conventional wisdom is that latency is the dominant performance bottleneck in DSM machines. We show, however, that controller occupancy also has a substantial impact on application performance, for the occupancies that are being proposed for a range of cache-coherent DSM machines. Moreover, performance is affected more by the contention that controller occupancy induces than the latency it adds. As expected, techniques to reduce the impact of network latency make controller occupancy a greater bottleneck. What is surprising, however, is that the performance impact of occupancy is substantial even for highly tuned applications and even in the absence of latency hiding techniques. Scaling the problem size is often used as a technique to overcome limitations in communication latency and bandwidth. We show that in many structured computations occupancy-induced contention is not alleviated by increasing problem size, and that there are important classes of applications for which the performance lost by using higher latency networks or higher occupancy controllers cannot be regained easily, if at all, by scaling the problem size.

1 Introduction

Distributed shared memory (DSM) multiprocessors are converging to a family of architectures that resemble the generic system shown in Figure 1.1. This architecture consists of a number of processing nodes connected by a general interconnection network. Each node contains a processor, its cache subsystem, and a portion of the total main memory of the machine. The processing node also contains a *communication controller,* which is responsible for managing the communication between its node and other nodes. Our interest in this paper is in the specific class of cache-coherent DSM machines, which provide communication and coherence at cache-line granularity.

There are many ways to build cache-coherent DSM machines, arising from differences in desired performance and cost characteristics and the extent to which a design uses commodity parts rather than customized hardware. In keeping with current trends, we assume the use of a commodity microprocessor, cache subsystem, and main memory. The major sources of variability are in the network and in the communication controller, which together constitute the communication architecture of the multiprocessor.

Candidate networks vary in their latency and bandwidth characteristics as well as in their topologies. They range from the low-latency, high-bandwidth networks used in massively parallel processors (MPPs), all the way to commodity local area net-

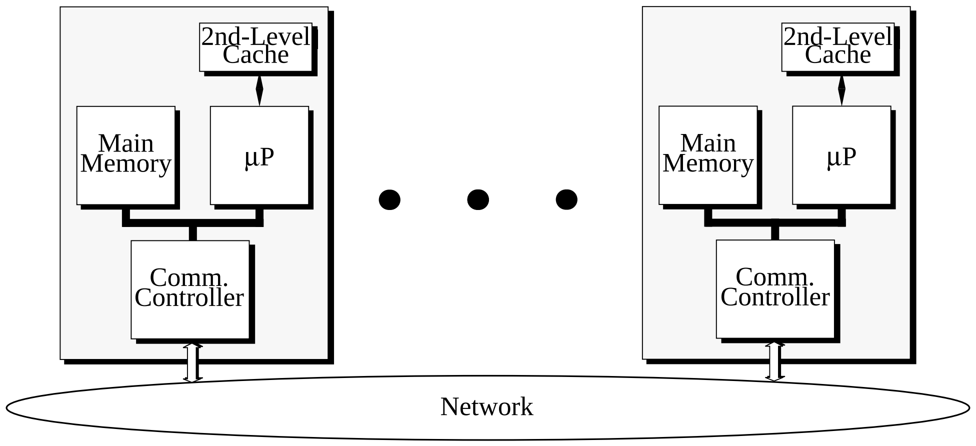


Figure 1.1. Generic DSM architecture

works (LANs). On the controller side, there are two important and related variables. One is the location where the communication controller is integrated into the processing node. This can be the cache controller, the memory subsystem, or the I/O bus. The other design variable is how customized the communication controller is for the tasks it performs; for instance, it may be a hardware finite state machine, a special-purpose processor that runs protocol code in response to communication-related events, or an inexpensive general-purpose processor.

Because of their differences in design cost and design effort, all of these types of systems are interesting. Current and proposed architectures for cache-coherent distributed shared memory have different design goals along these axes, and thus there are examples of real machines or proposed machines at almost every point in this design space. The question we address in this paper is how the characteristics of the network and controller affect the performance of parallel programs written for these machines. That is, as we move from more tightly-integrated and specialized communication architectures to less tightly-integrated and more commodity-based systems, how significant is the loss in parallel performance over a range of computations? We address this question by studying a range of important computations and communication architectures through a combination of detailed simulation and analytical modeling.

We characterize the communication architectures of DSM multiprocessors by a few key parameters that are similar to those in the *logP* model [CKP+93]. More detailed design decisions and machine parameters, which are crucial to establishing an architectural context, are held constant in our evaluation. We describe this architectural context in the next section. Section 3 describes the framework and methodology we use to study the effectiveness of different types of DSM architectures. Section 4 and Section 5 present and analyze our results, and Section 6 concludes the paper.

2 Architectural Context

Using the *logP* model, we abstract the communication architecture of a parallel machine in terms of four parameters. The *l* parameter in the *logP* model is the network latency from the moment the first bits of a message enter the network at a source node to the moment the first bit of the message arrives at the destination node, *o* is the overhead of sending a message, *g* is the gap (reciprocal of node-to-network bandwidth), and *P* is the number of processors. The only difference between our model and *logP* is in the *o* parameter. In *logP*, the overhead, *o*, is the time during which the main processor is busy initiating or receiving a message and cannot do anything else. In most DSM machines, however, protocol processing is off-loaded to a separate communication controller, and the main processor is free to continue doing independent work while the controller is occupied. The *o* parameter in our DSM model, then, stands for the *occupancy* of the communication controller per protocol action or message; that is, the time for which the controller is tied up with one action and cannot perform another. Alternatively, occupancy can be viewed as the reciprocal of the communication controller's message bandwidth. Since controller bandwidth may be confused with (the very different) network bandwidth parameter, however, we prefer to use the term controller occupancy.

We fix the number of processors, *P*, at 64 in this paper, so that we are studying moderate-scale DSM machines. Each processor is 200 MHz statically-scheduled dual-issue superscalar, for which we assume an effective IPC of 1.5 (without memory effects). The other three parameters that characterize the communication architecture—latency, occupancy, and bandwidth or gap—all have complicated aspects to them, and we make certain simplifying assumptions. Let us discuss each parameter individually, before setting the range in which we vary these parameters.

Latency: The latency of a message through the network depends, among other things, on how many hops the message travels in the network. For the moderate-scale machines that we consider, the overhead of getting the message from the processor into the network and vice versa dominates the topology-related component of the end-to-end latency seen by the processor. We therefore ignore topology, and compute network latency as the average network transit time between two nodes in the network. The average network transit time is defined to be the average time to traverse an 8 x 8 mesh, assuming no network contention.

Occupancy: The occupancy that the controller incurs for a request affects performance in two ways. First, it contributes directly to the end-to-end latency of the current request because the request must pass through the controller. Second, it can contribute indirectly to the end-to-end latencies of subsequent requests, through contention for the occupied controller. Occupancy is more difficult to represent as an abstract parameter than network latency for two reasons. First, we have to decide which types of transactions invoke actions on the controller and how much occupancy they incur. Second, the occupancy of a remote miss is actually distributed between two (or three) of the controllers in the system, and the occupancies of each of those individual transactions may not be the same. While we would like to represent occupancy by a single parameter *o*, occupancy in real machines often depends on the type of the transaction. Let us examine these two modeling challenges separately.

Clearly, all events related to internode communication and protocol processing incur controller occupancy. These events include cache misses that need data from another node, processor references that require the communication of state changes to other nodes, and incoming requests and replies from the network containing data and protocol information. We assume that cache misses that access local memory and do not generate any communication do not invoke the controller and thus incur no occupancy [SFL+94]. We also assume that the state lookup that determines if a cache miss needs to invoke the controller is free, and we assume uniprocessor nodes so that the communication controller has to handle the requests of only one local processor. All of these assumptions minimize the burden on the communication controller and hence expose more fundamental limitations. Machines with multiple processors per node and machines where the controller handles local memory references may perform worse than the results presented in this paper for the same values of controller occupancy, indicating that for some architectures controller occupancy may be even more important than we will show it to be.

In many machines, particularly those in which the communication controller runs software code sequences for protocol processing, the occupancies of the controller are different for different types of protocol actions. We make the following assumptions about occupancy. When the communication controller is simply generating a request into the network or receiving a reply from the network it incurs occupancy o. When the communication controller is the home of a network request it incurs occupancy $2o$, because it has to retrieve data from memory and/or manipulate coherence state information [HKO+94]. In this case we assume the memory access happens in parallel with the operation of the controller. If the state lookup at the home reveals that the requested line is dirty in the home node's cache, the communication controller incurs an extra fixed occupancy C while retrieving the data from the processor's cache. If the requested line is dirty in a third processor's cache, the home node incurs an occupancy of $2o$ and forwards the request to that processor, and the communication controller at that node incurs an occupancy of $2o+C$. The only other time occupancy is incurred is when the communication controller at the home node services a write request and sends invalidations to all nodes that are sharing the data. In this case the controller incurs an additional occupancy of two processor clocks per invalidation that it sends.

Bandwidth or **gap**: The gap (g) parameter specifies the node-to-network bandwidth. It determines how fast data can be transferred through the network interface, i.e. between the communication controller and the network itself. While we explicitly vary l and o in this paper, we do not vary node-to-network bandwidth. Instead, we fix it at 400 MB/s peak, which corresponds to MPP networks on current-generation machines. We compute the node-to-network bandwidth requirements of the individual applications in Section 4.3 and we show that a bandwidth of 400 MB/s is large enough to never be the performance bottleneck in our experiments. For coherence messages that do not carry data, the occupancy of the communication controller always dominates this gap limitation. For messages that carry data, this gap can theoretically become the bottleneck before controller occupancy for the two lowest occupancies we examine. We never observed the symptomatic filling of network interface buffers, however, both because our processor model allows only a limited number of outstanding requests, and because transactions that do not involve data are usually interspersed with those that do.

Given these assumptions about l, o and, g, let us examine the path and cost of a read miss to a cache line that is allocated on a remote node and is clean at its home. The request travels through the communication controller on the requesting node (o), traverses the network (l), travels through the communication controller at the home where the request is satisfied ($2o$), traverses the network again (l), and finally travels back through the communication controller at the source node (o). Including the fixed external interface delays into and out of each controller (K_{in}, K_{out}). This request has a total round-trip latency (without any contention) of $K_{in} + o + K_{out} + l + K_{in} + 2o + K_{out} + l + K_{in} + o + K_{out}$, or $2l + 4o + 6K$ if we assume $K_{in} = K_{out} = K$. If the line were dirty in the home node's cache, there would be an extra fixed cost of C at the home for retrieving the data from the cache. For a line that is dirty in the cache of a third processor (not the requestor or the home), the latency would be $3l + 6o + C + 8K$.

Table 2.1. DSM Design Space

Controller Occupancy (200 MHz Processor Clocks)	**Network Latency (200 MHz Processor Clocks)**			
	Tightly-Coupled MPP l=50 Low Latency	Distributed MPP l=100, 200, 400 Medium Latency	LANs: Aggressive ATM l=800 High Latency	LANs: Current ATM l=1600 Very high Latency
Hardwired Controller $o \leq 14$ pclocks, Low Occupancy	L1 O1	L2, L4, L8 O1	L16 O1	L32 O1
Customized Co-processor $28 \leq o \leq 56$ pclocks, Medium Occupancy	L1 O2, O4	L2, L4, L8 O2, O4	L16 O2, O4	L32 O2, O4
General-purpose Co-processor on Memory Bus o=112 pclocks, High Occupancy	L1 O8	L2, L4, L8 O8	L16 O8	L32 O8
General-purpose Co-processor on I/O Bus o=224 pclocks, Very high Occupancy	L1 O16	L2, L4, L8 O16	L16 O16	L32 O16

The network latency *l* and the controller occupancy *o* are the variables in the above times. We study a range of values for each variable, as shown in Table 2.1, covering a variety of interesting architectural alternatives. Our latencies *l* vary from tightly-coupled, low-latency MPP networks, through physically distributed MPP networks, all the way to commodity LANs composed of ATM switches. Although our highest value of latency corresponds to the latency of an ATM switch, it does not represent actual ATM networks because the 400 MB/s bandwidth we assume is much higher than current ATM bandwidth. Small values of occupancy represent communication controllers which are tightly-integrated, hardwired state machines. Such controllers appear in MIT's Alewife machine [ABC+95], the KSR1 machine [KSR92], the Stanford DASH multiprocessor [LLG+92], and Silicon Graphics's Origin 2000 [LLA+97]. As *o* increases the controller becomes less hardwired and more general-purpose, from specialized co-processors like those in the Stanford FLASH multiprocessor [KOH+94] the Sun S3.mp [NAB+95], and the Sequent NUMA-Q [LC96], through inexpensive off-the-shelf processors on the memory bus as in Typhoon-1 [RPW96], to a controller on the I/O bus of the main processor like those in SHRIMP [BLA+94], and the IBM SP2 [SSA+95]. The entries in Table 2.1 correspond to specific values in our range of latencies and occupancies, which like all cycle times in this paper are expressed in 200 MHz processor clocks (pclocks). The L1, O1 point is our base architecture, corresponding to a machine with a tightly-coupled network (*l*=50 pclocks) and a hardwired communication controller (*o*=14 pclocks). The other entries in the table correspond to the cross product of machines obtained by doubling network latency (L2=100, L4=200, L8=400, L16=800, L32=1600 pclocks) and doubling controller occupancy (O2=28, O4=56, O8=112, O16=224 pclocks). Each table entry or controller-network combination represents a potentially viable architecture based on cost/performance tradeoffs, including each of the examples mentioned earlier in this paragraph. The absolute range of controller occupancies studied is less than the range of network latencies because most current and proposed machines fall within the O1 to O16 range. However, as we shall see, using higher occupancy controllers surprisingly can have a more substantial effect on performance than using higher latency networks.

With this context established, we now present our framework for studying the effects of varying *l* and *o* on system performance over a range of important parallel computations.

3 Framework and Methodology

3.1 How We Approach the Problem

Our goal is to understand the impact of latency and occupancy on the effectiveness of a DSM system, and thus evaluate how much performance we lose as we relax the aggressiveness of our communication architectures along these dimensions. The first question we must resolve is how to evaluate this. We measure parallel performance as parallel efficiency, which is the speedup over a sequential implementation on a uniprocessor execution divided by the number of processors used. One possibility, then, is simply to examine how performance on a fixed problem size degrades as *l* and/or *o* are increased. This provides useful insights, and may be the most important question to a user who cares about that particular problem size. While this metric does provide useful insights and we use it for part of our results, it is not sufficient since for a given number of processors, most sources of reduced parallel efficiency, including the ratio of communication to computation and the load imbalance, depend strongly on the problem size that is used. Typically, sources of overhead like the communication to computation ratio decrease with increasing problem size. In fact, even the nature of the dominant overhead—e.g. whether it is communication or load imbalance—can change with problem size. We therefore believe that the best way to cast the effectiveness question is: *Given an application, a number of processors, and values for l and o that characterize the network and controller, what is the minimum sized problem that can deliver a desired level of parallel efficiency?*

The remaining issue is the choice of the "desirable" parallel efficiency. Typically, the larger the desired efficiency the larger the problem size needed for a given combination of *l* and *o*. The efficiency level we choose is therefore an important determinant of the constant factors in the expression for the required problem size. The desirable efficiency level may also change the relative importance of different performance bottlenecks, and hence the growth *rate* of the required problem size with *l* and *o*. In most cases, however, if the dominant bottleneck does not change, then the chosen level of efficiency will not affect the growth rate of the required problem size but only the starting point. We therefore use a desired parallel efficiency of 60% in this paper in an effort to set a fairly aggressive yet achievable performance target.

Some machine designers argue that cost-performance is the best overall figure of merit [WH95]. Though this may be an important factor in the decision to purchase machines, it is difficult to pinpoint the costs of machines at every point in our design space, especially as advances in technology cause the costs to change over time. Instead, we stick with a pure performance metric. If designers want to spend less money and get cheaper, slower networks, our results will still indicate the performance of shared memory programs running on those less aggressive architectures. In fact, cost can be factored in separately with our performance results to use cost-performance as a metric.

Finally, given the large communication latencies on DSM machines, it is natural to try to hide these latencies when possible. Latency can be hidden by various techniques, all of which exploit the availability of additional bandwidth and require that the processor allows multiple outstanding references. To hide write latency, we assume that the architecture supports a relaxed memory consistency model [cite Kourosh]. To hide read latency, we use software-controlled prefetching. Since read latencies are more difficult to hide, we use two versions of our applications where applicable: one that tries to hide read latency with programmer-inserted prefetches, and the other that does no prefetching at all.

3.2 Simulation Environment

The results presented in this paper are gathered from a detailed multi-threaded memory simulator that interfaces to the Tango Lite event-driven reference generator [Golds93]. The simulator models contention in detail within the communication controller, between the controller and its external interfaces, at main memory, and at the system bus. For the communication controller's internal queue sizes and fixed interface delays, we use the values from the Stanford FLASH multiprocessor design as being representative given current technology [HKO+94]. The input and output queue sizes in the controller's processor and network interfaces are uniformly set at 16 entries. We assume processor interface delays of 2 pclocks inbound and 8 pclocks outbound, and network interface delays of 16 pclocks inbound and 8 pclocks outbound. These constitute the K terms discussed in Section 2, although we note that K_{in} and K_{out} are not equal. The total round trip interface delay encountered on a remote clean miss is 58 pclocks and is assumed to remain fixed as controller and network characteristics are varied. We also fix the access time of main memory at 140 ns (28 pclocks), a fairly aggressive number. Fixing the interface delays and the memory access time is realistic, and allows us to focus on the performance of the communication architecture.

The processor controls its own cache, and we assume that it takes 30 pclocks for the communication controller to retrieve state information and data from that cache when necessary. This is the value of C used in our simulations (see Section 2), and is also obtained from the Stanford FLASH multiprocessor. The caches are 1 MB in size, two-way set associative, and have a line size of 128 bytes. We also assume that the processor has both prefetch and prefetch exclusive instructions. In our processor model a load miss stalls the processor until the first double-word of data is returned, while prefetch, prefetch exclusive, and store misses will not stall the processor unless there are already references outstanding to four different cache lines. While this upper bound of only four outstanding cache lines can limit the amount of latency that a processor can hide with bandwidth, it is nonetheless more aggressive than the current situation in most commodity microprocessors.

Table 3.1. Comparison of Application Speedups Between the Stanford DASH Multiprocessor and the Simulator

	16 processors		32 processors	
Application	**DASH**	**Simulator**	**DASH**	**Simulator**
Barnes (16K particles)	11.6	11.1	22.4	22.1
Ocean (514x514 grid)	10.3	9.5	19.4	18.8
Radix (4M keys)	13.1	11.8	26.4	23.5
FFT (1M points)	11.6	11.2	19.2	21.5

Validation: To validate our simulator, we compared the simulator to an existing moderate-scale shared memory multiprocessor, the Stanford DASH machine [LLG+92]. We use a 32 processor DASH system, arranged in 8 nodes connected by a mesh network topology. Each DASH node is a cluster of 4 processors sharing a common memory bus. The CPUs run at 33 MHz, and the system bus is 16 MHz. The primary instruction and data caches are 64 KB each, with a unified 256 KB direct-mapped secondary cache. Our simulator does not support clustering, but we modified its other parameters to match the architectural characteristics of DASH. The results of the comparison are shown in Table 3.1. In each case, the speedups predicted by the simulator were within 12% of the actual values obtained on DASH, and within 5% for half of the trials. The simulated speedups are in fact usually lower than those of the real machine. This is encouraging from our perspective because it suggests our speedups are not necessarily optimistic. One reason why the simulated speedups tend to be low is that the simulator models only a large first-level cache which helps uniprocessor performance more than parallel performance. Although only 16 and 32 processor numbers are shown in the table, we compared smaller numbers of processors as well, with similar results.

3.3 Applications

The applications we use in our study are summarized in Table 3.2. They include three complete applications (Barnes-Hut, Ocean, and Water) and three computational kernels (FFT, LU, and Radix). The programs were chosen because they represent a variety of important scientific computations with different communication patterns and requirements. Barnes is representative of the class of hierarchical N-body methods, which are used in astrophysics, electrostatics, and plasma physics, among other domains. Ocean is representative of many computational fluid dynamics applications on regular grids, and uses a multi-grid equation solver. Water is representative of computational chemistry applications which compute particle interactions based on a cutoff radius. FFT forms the computational core of a variety of applications, including image and signal processing as well as climate modeling. We use an optimized, high-radix version of FFT. The most common need for large dense LU factorization is in radar cross-section problems; however, for our purposes dense LU factorization is very similar to more widely used sparse matrix factorization techniques (such as blocked Cholesky factorization [Roth93]), and of several other matrix factorization and eigenvalue methods. Finally, Radix is a widely used sorting algorithm. Descriptions of the applications can be found in: Barnes [HS94]; Radix and Ocean [WSH94]; Water [WOT+95]; FFT and LU [RSG93]. The applications are quite highly optimized to improve communication performance, and particularly to reduce spurious hot-spotting or contention effects that adversely impact controller occupancy. The codes for the applications are taken from the SPLASH-2 application suite [WOT+95], although Radix was modified to use a tree data structure (rather than a linear key chain) to communicate ranks and densities efficiently.

Table 3.2. Applications and Their Communication Patterns

Application	Description	Communication Pattern
Barnes	Barnes-Hut hierarchical N-body simulation	irregular, hierarchical
Ocean	Multigrid large scale ocean simulation	nearest neighbor iterative, hierarchical
Water	Molecular dynamics simulation	structured, many-to-many
FFT	Radix $\sqrt{n}$ Six-Step Fast Fourier Transform	regular, all-to-all, blocked matrix transpose
LU	Blocked dense LU decomposition	structured, one-to-many
Radix	Integer radix sort	irregular, all-to-all

4 Results for a Fixed Problem Size

First, it is useful to examine how the parallel efficiency of an application changes with *l* and *o* for the problem size that yields the desired level of efficiency on our most aggressive architecture. This lends important insight into how latency and occupancy interact, and how much each contributes to performance degradation. It also helps to guide our search for the larger problem sizes needed to retain the desired efficiency.

4.1 Intuition

As *l* and *o* increase for a given problem size, parallel efficiency clearly should decrease. But can we predict *how* it decreases for the different applications? Excluding load imbalance and the time spent waiting at synchronization points, the time taken by a parallel application can be broken down into two components: local computation, including cache and local memory accesses, and communication. Communication cost can be further broken down into the cost due to round-trip latency and the cost added by contention for a resource of limited bandwidth (non-zero occupancy). If we assume perfect load balancing of both computation and communication, and that local caching effects stay the same regardless of the number of processors used, then parallel efficiency is determined by the following formula:

$$\frac{T_{comp}}{T_{comp} + V_{comm} \times (T_L + T_C)} \tag{4.1}$$

where T_{comp} is the uniprocessor computation time, V_{comm} is the total volume of communication (number of communication misses incurred on all processors), and T_L and T_C are the average stall times due to latency and contention, respectively, for each communication.

To compare the effects of increases in controller occupancy with those of increases in network latency, we define T_L as the *communication latency*, which is the round-trip overall latency, assuming no contention, for a remote miss that is satisfied by the main memory of the home node (computed as $2l+4o+6k$ in Section 2). For a fixed problem size and number of processors, both T_{comp} and V_{comm} are constant. T_L varies linearly with *l* and *o*. The remaining question is how the contention component, T_C, varies with *l* and *o*. If controller occupancy does not cause any contention and contributes only to end-to-end latency, then $T_C = 0$ in Eq. 4.1, which means that a decrease in parallel efficiency caused by an increase in controller occupancy is indistinguishable from holding controller occupancy constant and making a corresponding increase in network latency. On the other hand, if higher occupancy causes contention for the controller, we would expect that a larger occupancy has a correspondingly larger value of T_C, and parallel efficiency would no longer be linearly dependent on *o*. Thus the parallel efficiency for a larger occupancy would be less than for a smaller occupancy, even if the overall communication latency T_L is the same in both cases.

To understand the performance impact of *l* and *o*, in this section we seek answers to the following questions: (i) starting from the base L1, O1 architecture (L1 = MPP network, O1 = hardwired controller), how does increasing network latency degrade performance for the base problem size, both with and without prefetching; and (ii) to what extent does controller occupancy cause contention in addition to contributing latency, both with and without prefetching, and how does this contention affect parallel efficiencies. Other interesting questions that we answer in the process are: What are the problem sizes needed to obtain 60% parallel efficiency for our applications on the base architecture, which represents an aggressive current-generation multiprocessor; and what are the node-to-network bandwidth requirements for these problem sizes?

4.2 A Case Study

In this section we outline the framework we use to discuss our fixed problem size results in the context of a single application. We choose FFT because it is a simple, well-understood program. For both the prefetched and non-prefetched versions of the program, we obtain results for varying latency and occupancy through detailed simulation. Many of the important insights emerge in the course of this case study. In Section 4.3 we present results for all our applications, and compare them with those obtained for FFT.

The results presented here are graphs of parallel efficiency vs. communication latency for both non-prefetched and prefetched versions of FFT. Each graph shows a family of curves, one for each value of occupancy *o*, ranging from O1 (a hardwired controller on the memory bus) to O16 (a general purpose processor on the I/O bus). Each curve shows how parallel efficiency changes with increasing communication latency ($T_L = 2l+4o+6k$). The points along each curve show network latency *l* increasing by factors of two from L1 (tightly-coupled MPP latency) to L32 (current ATM network latency). All graphs in this paper have this same format. If there is no contention at the controllers ($T_C = 0$), then with T_L as the x-axis we see from Eq. 4.1 that the curves for different values of *o* should be identical. Different curves for different values of *o* indicate occupancy-induced contention.

Without Prefetching: Figure 4.1(a) graphs parallel efficiency vs. communication latency for our base FFT problem size (256K points). For FFT, the required problem size for 60% efficiency on 64 processors is small enough in our most aggressive architecture that artifacts of its interactions with processor cache line size throw it off the growth rate curve. We therefore use an efficiency of 80% for FFT, though the growth rates would have been the same if we had used 60% with a smaller cache line size that did not cause these artifacts. The first interesting result is that varying controller occupancies generates multiple efficiency curves, indicating that the contention component of occupancy is indeed important, even without prefetching and even though the program is optimized to reduce contention. The curves also begin to flatten as *o* is increased, which indicates that the controller starts to saturate, and its high utilization becomes the performance bottleneck in the machine.

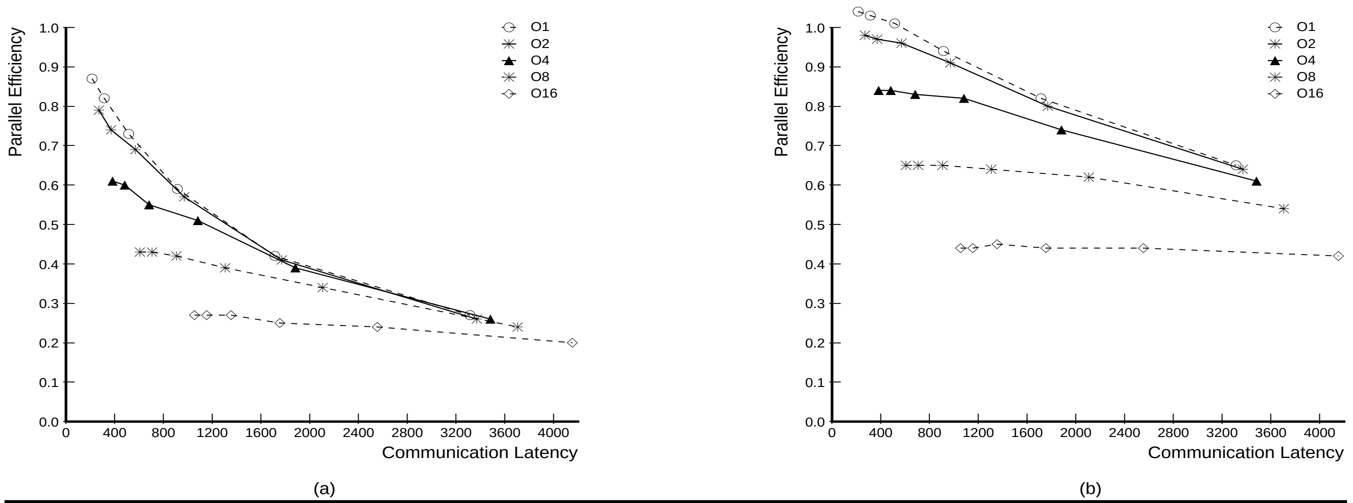


Figure 4.1. FFT base problem size results for both the (a) non-prefetched and (b) prefetched versions

Note that the parallel efficiencies nearly converge at high values of *l*, implying that at today's ATM latencies, controller occupancy does not have a large impact on overall performance for this problem size without prefetching. Conversely, for a range of MPP and distributed MPP network latencies (small values of *l*), controller occupancy is a critical determinant of overall performance. What may be surprising are the values of controller occupancy at which the efficiency curves begin to diverge at low *l*. At the smallest value of network latency (L1), a hardwired controller (O1) is 1.10 times to 1.43 times faster than a customized co-processor (O2 to O4). The slowest customized co-processor is in turn 1.42 times faster than a general-purpose processor on the memory bus (O8), and 2.26 times faster than a general-purpose processor on the I/O bus (O16). Not only is T_c

non-zero, it increases as the controller occupancy *o* increases. Furthermore, it is increases in T_c that account for the efficiency lost while communication latency is held constant and controller occupancy is increased.

With Prefetching: Figure 4.1(b) shows that with prefetching, there are also multiple efficiency curves ($T_C > 0$) that flatten out as *o* increases. Unlike the non-prefetched case, the efficiencies no longer converge because the contention component of occupancy affects overall performance even at high network latencies. Prefetching improves performance more at low *o* and up to moderate *l* than it does at higher values of *o* and *l*. Beyond a certain *l*, we cannot hide all the network latency, and increases in latency begin to hurt the prefetched case at the same rate as the non-prefetched case, causing the efficiency curves to take on similar shapes from this point on. At medium *o*, the controller becomes a bottleneck, as it is unable to match the increased bandwidth needs of prefetching. With the fastest MPP networks we use, a hardwired controller (O1) is 1.60 times faster than a general purpose processor on the memory bus (O8), and 2.36 times faster than a general purpose processor on the I/O bus. To support prefetching in DSM machines then, it is crucial to keep the occupancy of the controller low.

Discussion: Let us look at the graphs from the perspective of an architect. Starting at today's ATM network latencies (L32), if we could reduce our network latency in half to reach the current goal of ATM networks, how much performance would we gain? The answer here depends on the occupancy of our controller and whether or not latency-hiding techniques are used. In the non-prefetched case a machine with a hardwired controller (O1) shows a 1.56 times improvement in parallel efficiency as network latency is halved from L32 to L16, while a machine with an O16 controller makes a 1.20 times improvement. In the prefetched case the improvement is 1.26 with an O1 controller and 1.05 with an O16 controller. While the relative gains in performance can be great, the absolute performance of both of these systems is still low compared to the base L1, O1 architecture.

Now suppose we start from the most efficient L1, O1 machine, and see how much we lose by relaxing each parameter. Beyond a very efficient customized controller on the memory bus (O2), controller occupancy is crucial to performance both with and without prefetching. For low occupancy controllers, going to a higher-latency network (say, doubling network latency) also hurts performance significantly, though with prefetching the performance impact is smaller. Once the controller is a general-purpose processor on the memory or I/O bus (an O8 or O16 machine), doubling network latency does not significantly affect performance. Surprisingly, starting from a very efficient L1, O1 or L1, O2 machine, doubling controller occupancy hurts performance more than doubling network latency (in the prefetched case the base architecture is 1.06 times faster versus 1.01 times faster, respectively), even though the communication latency of the occupancy doubled machine is *less* than that of the latency doubled machine. As designers of tightly-coupled machines, if the cost considerations for doubling the two parameters are similar, we might favor keeping occupancy low and sacrificing some network latency.

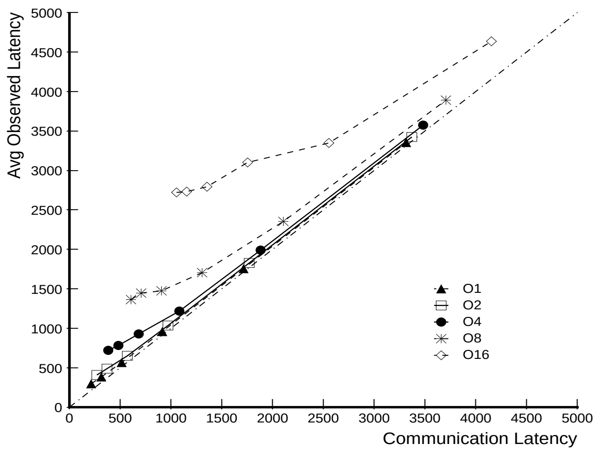


Figure 4.2. Observed remote read miss latency versus Communication Latency for non-prefetched FFT

Another way to view the impact of contention on performance is to compare, for different controller occupancies, the average observed remote read miss latency with the expected uncontended remote read miss latency. These results are shown for non-prefetched FFT in Figure 4.2. The observed latency is taken from the simulation results, while the expected latency is simply the communication latency as defined in Section 2 ($2l + 4o + 6K$ in most cases). The observed latencies of the O1 and O2 architectures are almost identical to their corresponding communication latencies. For all controller occupancies except O16, the observed latency converges to the communication latency at the highest network latencies, as expected. But starting at an O8 controller, the observed latency becomes noticeably larger than the communication latency at all network latencies. For example, an L1, O16 machine has an observed latency which is over 2.5 times slower than the expected read miss latency.

We found that for FFT, without prefetching, controller occupancy is critical at low network latencies but not at high latencies, while with prefetching it is critical at all latencies. Since high-latency networks make it all the more important to hide latency, occupancy is in effect critical at all values of network latency. The trend toward lower latency networks further increases the impact of occupancy. An interesting result is that it is usually the contention component of controller occupancy (T_c), not its latency component (its contribution to T_L), that dominates its contribution to performance degradation, *both with and without prefetching*. We find that for most of our applications, for controller occupancies above O2—which represents an efficient, customized co-processor—70%-95% of the performance degradation due to increasing occupancy is attributed solely to its contention component for almost all values of network latency.

4.3 Modeling

A natural question is if it is possible to generate the efficiency curves presented in Section 4.2 using queueing models, rather than relying on simulation results, especially for an analytically describable application like FFT. We address this question by developing an application independent queueing model for parallel efficiency based on Eq. 4.1. Recall from Eq. 4.1 that parallel efficiency is determined by four parameters: T_{comp}, the uniprocessor computation time, V_{comm}, the number of communication misses, T_L, the component of the average remote miss time due to latency, and T_C, the component of the average remote miss time due to contention. Let us look at how we model each of these parameters.

$\boldsymbol{T_{comp}}$: Since T_{comp} is just the uniprocessor computation time, it remains constant across different communication architectures. To determine T_{comp} we use the same simulated uniprocessor time we use to calculate parallel efficiency.

$\boldsymbol{V_{comm}}$: Without prefetching, V_{comm}, the number of remote misses, also remains constant across different communication architectures. Since the problem sizes for each application and the cache configuration for each machine are also fixed in Section 4.2, we simply extracted the number of remote read misses from the simulation results for the L1, O1 machine. In our model, we assume that the latency of remote writes is completely hidden by the write buffer, and therefore writes do not affect parallel efficiency. Modeling the effects of prefetching across different communication architectures is much less accurate than assuming no prefetching, so we model only the non-prefetched versions of the applications.

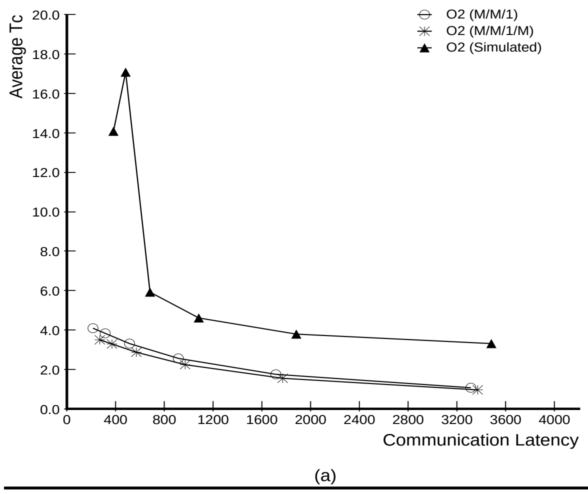
$\boldsymbol{T_L}$: This is the easiest parameter to model. We simply compute T_L using the $2l + 4o + 6K$ formula developed in Section 2. If the misses are expected to be dirty at the home, T_L is $2l + 4o + 6K + C$, and if the misses are dirty remote, then a T_L of $3l + 6o + 8K + C$ is used.

$\boldsymbol{T_C}$: As we saw in Section 4.2, the amount of controller-induced contention in these applications is both significant and highly dependent on the underlying communication architecture. To model the amount of contention in the system, we treat each communication controller as a queueing system. Messages arrive from either the network or the processor, queue up, and are serviced by the controller. Since messages do not experience contention while in the network, but only entering and leaving the network, T_C is measured by the amount of time spent in the queue for the controller. We also make the assumption that the number and types of messages seen by each controller are the same. This is a reasonable assumption provided both the workload and data distribution are balanced across processors (true in FFT which we are presenting results for here), and allows us to focus on modeling only one controller, calculating the average time spent in its queue, and using that number as the value of T_C for all processors.

We implemented two different queueing models: one which assumes an infinite controller input queue (M/M/1), and one which assumes that a maximum number of messages can be queued at the communication controller (M/M/1/M), depending on the size of the write buffer. The service time for both models is the average occupancy incurred during a read miss, determined by extracting the number and types of protocol messages issued during a simulation of the particular machine configuration, and computing a weighted average of their occupancies. The average arrival rate is the number of messages that arrive at the controller divided by the parallel execution time, assuming no contention. Note that the arrival rate includes all the messages that the controller must execute to handle the cache coherence protocol, not just those specifically needed to handle the remote read misses.

The predicted T_C vs. communication latency for both models is shown in Figure 4.3 along with the T_C determined through simulation, for FFT with an O2 controller (a), and then with an O8 controller (b). Clearly, neither queueing model predicts T_C very well; the models are off by as much as an order of magnitude. The main reason why the models are so far off is that a constant, uniform arrival rate is an invalid assumption. Unfortunately it is very difficult to model what the proper arrival rate should be, since it depends greatly on the amount of contention in the system.

We found that we could not accurately generate the efficiency curves in Section 4.2 using application independent simple queueing models. While it is possible to accurately model portions of the total execution time, modeling the amount of contention present in the system proved difficult. In order to develop an accurate model for non-prefetched FFT, the exact communication structure must be understood, both in terms of the sequence and number of messages. Although it is possible to determine this information for FFT, it is much harder for the other, less easily analyzed applications. Therefore, in the next sec-



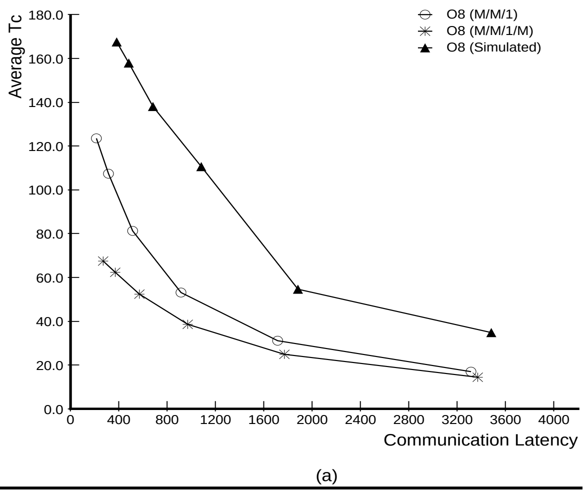


Figure 4.3. Modeled versus simulated T_C for FFT with (a) an O2 controller and (b) an O8 controller

tion we continue to present simulation results for the other applications as the only means of accurately presenting the impact of controller occupancy on application performance.

4.4 Simulation Results for Other Applications

For each of the applications, Table 4.1 first shows the minimum problem size needed to achieve 60% efficiency on the most efficient L1, O1 architecture. All of the problem sizes have small data set sizes. The table also shows the per-processor communication bandwidth requirements for that problem size (including all protocol messages). The bandwidth numbers are presented in megabytes divided by total execution time, not just that of the communication phases. Note that the bandwidth numbers in Table 4.1 are moderate, and in all cases are much less than our node-to-network bandwidth of 400 MB/s. Even with burstiness, we find network bandwidth not to be a bottleneck, as mentioned earlier. Bandwidth requirements will generally decrease as problem sizes are increased. Following the framework developed in Section 4.2, we now present base problem size results for all of our applications. We compare these results with those we have already seen for our FFT case study.

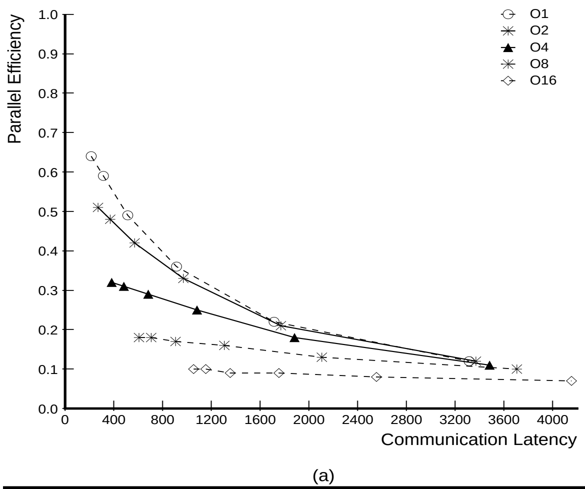
Table 4.1. Minimum Problem Sizes and Per-Processor Bandwidth Requirements for the Base Architecture

	Non-prefetched		Prefetched	
Application	**Minimum Problem Size**	**Node Bandwidth (MB/s)**	**Minimum Problem Size**	**Node Bandwidth (MB/s)**
Barnes	8192 particles	7.4	N/A	N/A
Ocean	258x258 grid	38.6	258x258 grid	44.8
Water	512 molecules	10.9	N/A	N/A
FFT	64K points	47.2	64K points	60.0
LU	512x512 matrix	7.5	512x512 matrix	7.9
Radix	2M keys, radix 256	70.8	1M keys, radix 256	79.6

Radix: The results for Radix shown in Figure 4.4 are similar to those for FFT, with a few notable exceptions. Like FFT, without prefetching all the parallel efficiencies almost converge by today’s ATM latencies (our rightmost points). While the O1 and O2 efficiency curves are still very close together, the O8 curve is much flatter than it is in FFT, and the O16 curve is almost totally flat. This indicates that in Radix contention for the controller matters even more than it does in FFT.

In the prefetched version of Radix, we again see a linearization of all the curves, although prefetching is not as successful in Radix as it is in FFT, because Radix communicates mostly through writes. The key prefetching trends continue in Radix: prefetching helps much more at lower values of *o* and moderate *l*, and the curves do not converge to a point at L32, indicating that it is critical to keep occupancy low when prefetching, even with ATM latencies.

LU: The results for LU (Figure 4.5) are also similar to those for FFT. One significant difference for both prefetched and non-prefetched LU is that the performance is less sensitive to both latency and occupancy. The reason is that LU has a high



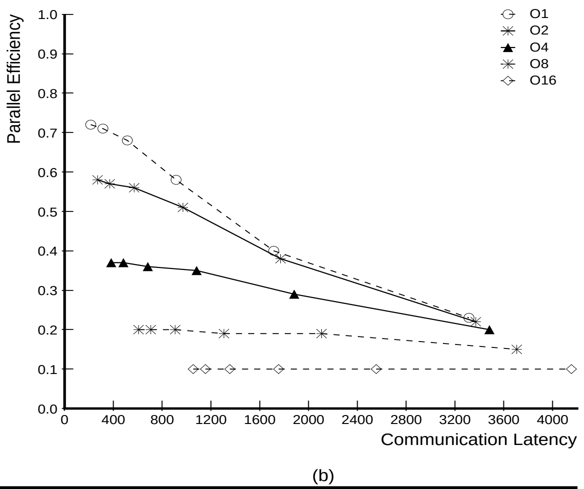
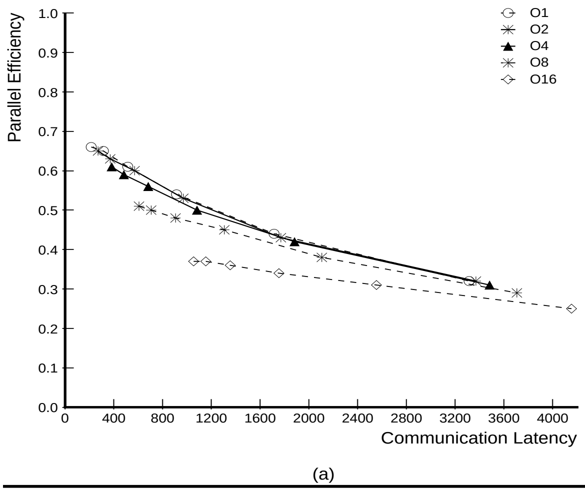


Figure 4.4. Radix results for the base problem size for both the (a) non-prefetched and (b) prefetched versions



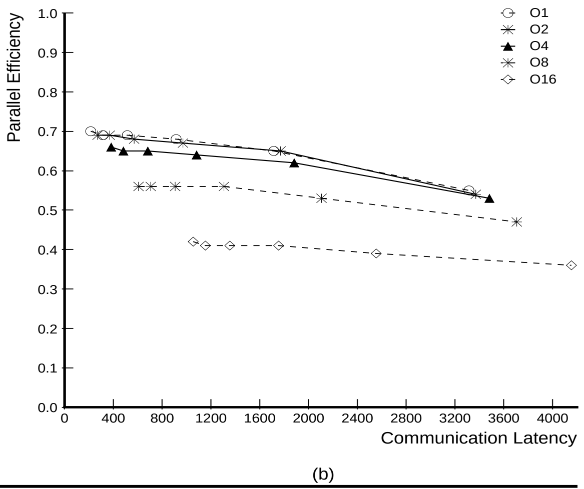


Figure 4.5. LU results for the base problem size for both the (a) non-prefetched and (b) prefetched versions

computation-to-communication ratio, and the dominant bottleneck is load imbalance, so its performance is less dependent on communication costs.

Ocean: Ocean, which performs many iterative nearest-neighbor computations on regular grids, including a multigrid solver, depends more on network latency than any of the previous applications. However, it depends substantially on controller occupancy as well (Figure 4.6). Unlike the previous applications, Ocean cannot take full advantage of spatial locality when it communicates data, leaving it especially sensitive to changes in network latency. Prefetched Ocean cannot hide enough of the latency, so the prefetched efficiency curves are also somewhat concave.

Barnes and Water: Figure 4.7 shows the results for Barnes and Water. Neither application includes prefetching, because the high degree of temporal locality (and irregularity in Barnes) makes it difficult to determine which particular memory references will miss in the cache, and prefetching all of the references that may cause communication incurs too much overhead. For Barnes, the O1 and O2 efficiency curves are almost identical. The O4 curve is different only for the lowest values of *l*, and the curves do not begin to diverge until O8. Again, the parallel efficiencies all converge at high network latency. Of all the applications, these two have the least performance variation across the design space. In particular, they are the least occupancy-bound of all the applications. However, they have very small cache miss rates, and performance is nonetheless impacted quite substantially by using O8 or O16 controllers for low latency MPP networks. For example, in Barnes the L1, O1 machine is 1.28 times faster than an L1, O8 machine, and 1.66 times faster than a L1, O16 machine.

As we expected, increasing network latency uniformly decreases overall performance across all the applications. Prefetching is often very effective at improving performance, especially at moderate latencies, but requires low occupancy controllers.

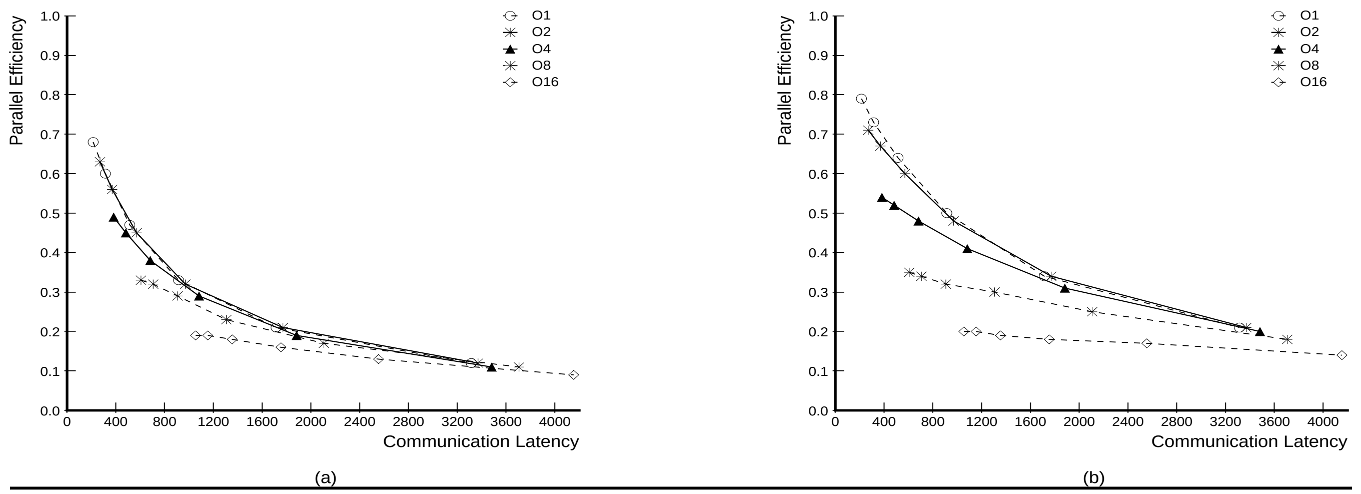


Figure 4.6. Ocean results for the base problem size for both the (a) non-prefetched and (b) prefetched versions

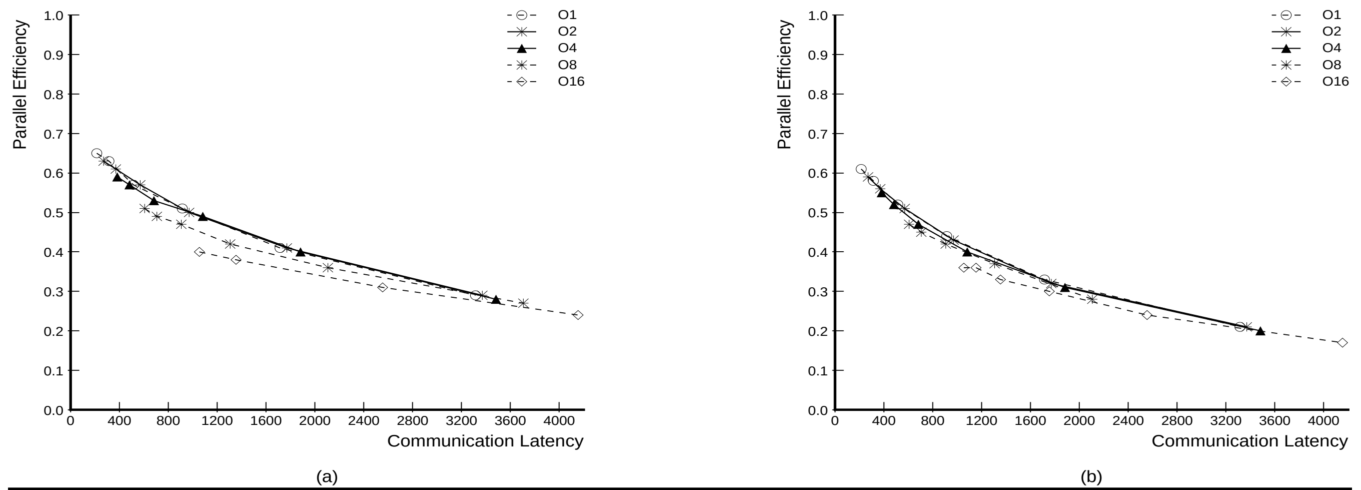


Figure 4.7. Base problem size results for (a) Barnes and (b) Water

It is also significant that even when prefetching, all of these applications have very low node-to-network bandwidth requirements. Though there are some differences in the applications, the surprising result is that consistently across all applications the contention component of controller occupancy has a significant performance effect. This effect is particularly acute at low values of network latency. In addition, the point at which the efficiency curves begin to flatten occurs at relatively small values of occupancy, typically either O4 or O8, and by O16 (communication controller on the I/O bus) the curves are almost flat. From a design standpoint, these results show that controller occupancy will become a bottleneck unless the communication controller is a hardwired or customized controller integrated on the memory bus of the main processor.

5 Results for Increasing Problem Size

In the previous section we saw how parallel efficiency changed as we varied both network latency and controller occupancy for the fixed, base problem size. In designing DSM machines with high network latencies or controller occupancies, designers hope that these machines can be made to run at high parallel efficiencies simply by running somewhat larger problems [GMB88]. The question is how the problem size must be scaled to maintain 60% efficiency, and at what point do these problem sizes become unrealistic. In the results we present, we define problem size as the size of the application's data set, since this is usually the main application parameter affecting parallel efficiency in these programs. In most real scientific applications the execution time grows more rapidly than the data set size for a variety of reasons [SHG93], and we shall comment on this as well.

Ideally, we would determine the minimum problem sizes for all machine configurations through simulation. Unfortunately, the problem sizes required for the less aggressive architectures are too large to determine via simulation. The alternative is to use models, but as we showed in Section 4.3 it is very difficult to predict performance using queueing models for these applications, because it is difficult to predict the amount of contention. However, it is possible to determine how the contention-related effects of these applications scale to larger problem sizes without using queueing models. Recall from Eq. 4.1 that to determine the problem size needed to attain a given efficiency, we need to know not only how the amount of computation and communication (T_{comp} and V_{comm}) scale with problem size, but how the latency and contention costs of the average communication (T_L and T_C) scale as well. Determining how T_{comp} and V_{comm} scale is usually straightforward, and clearly T_L does not vary with problem size. The only question is to determine how T_C scales with problem size. The hope when using a high-occupancy controller is that the contention component T_C decreases as problem size increases. In the next section, however, we will show that T_C is often independent of problem size by returning to our case study of FFT. While this is bad news for less aggressive communication architectures, it allows us to use the same value of T_C determined from simulation results from smaller problem sizes for the prediction of performance at larger problem sizes. Thus, while we may not be able to predict what T_C will be for a given communication architecture and application, once we determine it through simulation, it will remain reasonably constant as the problem size increases. After we examine FFT, we present the results of increasing problem size for all of our applications in Section 5.2.

5.1 Case Study

In FFT, the growth rate of computation T_{comp} with the number of points n (the problem size) is $O(n \log n)$. The growth rate of communication V_{comm} is $O(n)$. To maintain a fixed efficiency as the average communication cost T_L+T_C increases by a factor of s, we must increase the problem size at a rate that keeps the ratio of computation time to communication time constant. If T_C is not a function of problem size, then this simply means we must keep the computation to communication ratio ($\log n$) constant, which requires an increase in n by an exponential factor of 2^s.

Through simulation we gathered efficiency results for non-prefetched and prefetched FFT at two problem sizes: our base problem size and one that is four times larger. We look at the cross product of O1, O8, O16 and L1, L4, L16, which represent some realistic machine configurations (see Table 2.1). Increasing the problem by a factor of four did not increase the efficiency much, either with or without prefetching.

An important result is that at all controller occupancies, the average communication time (T_L + T_C) remains constant in both the non-prefetched and prefetched versions of FFT. Since the average latency component of communication, T_L, remains constant by definition, this means that the average contention component, T_C, does not decrease with an increase in problem size. For example, in FFT, the average read miss time was 352 processor cycles for a problem size of 2^{18} points, 330 cycles for 2^{20} points, and 340 cycles for 2^{22} points. Although this seems counter-intuitive given that the overall computation-to-communication ratio increases, there is a clear explanation. In many structured applications, communication is isolated in different phases from local computation. As a result, although the overall computation-to-communication ratio over the whole application increases with problem size, *within* the communication phases the ratio remains constant as problem size grows. Since contention depends on the rate or burstiness of communication, and that rate is independent of problem size, it follows that the contention is independent of problem size as well. Thus, FFT indeed requires an exponential increase in problem size to overcome the effects of increased network latency or controller occupancy. Higher controller occupancies cause more contention, increasing the value of the exponent substantially.

This insight, that T_L *and* T_C are independent of problem size, allows us to predict the required problem sizes for FFT as l and o change, as long as we know how T_L and T_C change with l and o for a fixed problem size. Since FFT communicates through reads, we can use the average remote read miss time from the simulation results to estimate T_L + T_C. The simulations also provide us with the constants for the computation time. Optimistically assuming perfect load balancing, both of computation and communication, Table 5.1 shows the minimum problem size needed to reach 60% efficiency for the nine selected combinations of l and o. Of these, we were able to simulate the required problem size for a controller occupancy of O1 and network latencies of L1 and L4. The other numbers listed in Table 5.1 are predicted values, although the trends and contention effects have been validated.

Without Prefetching: It is clear from the table that increasing network latency causes an exponential increase in the required problem size. The contention component of controller occupancy also has a big impact on the required problem size for FFT, even without prefetching. For example, if the O8 controller had the same contention component T_C as the O1 controller, but the communication latency corresponded to O8, the problem sizes for O8 in Table 5.1 would have been 16384 times smaller at L1 and 64 times smaller at L16. For O16, the problem sizes would have been 65536 times smaller at L1 and 1024 times smaller at L16.

Table 5.1. Minimum Problem Size Required for 60% Parallel Efficiency for both Non-Prefetched and Prefetched FFT

Controller Occupancy	Network Latency					
	Non-Prefetched			Prefetched		
	L1	L4	L16	L1	L4	L16
O1	2^{16}	2^{16}	2^{44}	2^{16}	2^{16}	2^{16}
O8	2^{34}	2^{36}	2^{60}	2^{18}	2^{18}	2^{20}
O16	2^{70}	2^{72}	2^{86}	2^{30}	2^{30}	2^{30}

With Prefetching: For the same *l* and *o*, the minimum problem size needed is much smaller than for the non-prefetched version, and depends much less on latency. However, once the latency becomes too large to be hidden, the growth rate is exponential in the amount that cannot be hidden. Contention still plays a critical role in determining the required problem size. With the same contention component of the O1 hardwired controller the O8 general purpose controller would only need problem sizes 4 to 16 times smaller than those listed in Table 5.1. The O16 controller is far worse off: It would need a problem 16384 times smaller.

For both versions of FFT, the problem size needed to achieve the desired efficiency at high controller occupancies is unreasonably large. The same is true of the non-prefetched version at high network latencies. Although FFT performs well with an aggressive communication architecture, compromising the aggressiveness of a communication architecture, then, makes it be extremely difficult to achieve high parallel efficiencies even by growing the problem size.

5.2 Results for Other Applications

Radix: It is much more difficult to retain good efficiency in Radix than in FFT because the overall computation-to-communication ratio, not just in the communication phase, is constant. This means that unless contention decreases, increasing the problem size should not increase the efficiency. We find that as the problem size increases, contention actually worsens first, and then levels off. Table 5.2 summarizes the results. Contention worsens because the dominant communication in Radix is through writes (in the permutation phase), and these writes are bursty. As the problem size approaches the total amount of cache memory in the machine, it becomes increasingly likely that a given write will cause a cache line to be written back to the home, which is usually remote. This in effect doubles the number of messages that the communication controller at the home has to handle in the same small amount of time. The situation is actually much worse, because the irregularity of the communication causes some nodes to become hot-spots as problem size increases and the hot-spotting is both more prevalent and worse as controller occupancy increases. Without prefetching, only the base architecture reached 60% efficiency for any problem size. Even with prefetching, only the O1 and O2 controllers manage to reach 60% efficiency with the fastest network, and only the O1 controller can sustain 60% efficiency when a distributed MPP network (L4) is used.

Table 5.2. Minimum Problem Size Required for 60% Parallel Efficiency for both Non-Prefetched and Prefetched Radix

Controller Occupancy	Network Latency					
	Non-Prefetched			Prefetched		
	L1	L4	L16	L1	L4	L16
O1	2M keys	*impossible*	*impossible*	1M keys	2M keys	*impossible*
O8	*impossible*	*impossible*	*impossible*	*impossible*	*impossible*	*impossible*
O16	*impossible*	*impossible*	*impossible*	*impossible*	*impossible*	*impossible*

LU: LU scales much better than either Radix or FFT. One reason is that the computation-to-communication ratio in LU grows linearly in the problem size ($O(n^3)$ computation versus $O(n^2)$ communication). LU therefore requires much smaller increases in problem size to reduce relative communication costs. The other reason is that the main bottleneck for LU on an L1, O1 machine is load imbalance and not communication. Increasing the problem size improves load balance quickly as well.

Table 5.3. Minimum Problem Size Required for 60% Parallel Efficiency for both Non-Prefetched and Prefetched LU

Controller Occupancy	Network Latency					
	Non-Prefetched			Prefetched		
	L1	L4	L16	L1	L4	L16
O1	464^2 (1.0x)	512^2 (1.2x)	832^2 (3.2x)	432^2 (1.0x)	432^2 (1.0x)	464^2 (1.2x)
O8	696^2 (2.3x)	768^2 (2.7x)	1024^2 (4.9x)	608^2 (2.0x)	608^2 (2.0x)	672^2 (2.4x)
O16	1024^2 (4.9x)	1024^2 (4.9x)	1536^2 (11x)	812^2 (3.5x)	812^2 (3.5x)	844^2 (3.8x)

Like FFT and Radix, LU also communicates data in structured phases that in themselves have a constant computation-to-communication ratio. Consequently, contention does not decrease with increasing problem size, allowing us to easily predict the required problem size for machines with larger latencies and occupancies. Table 5.3 summarizes the results. For each entry, the value in parentheses is the ratio of the required data set size to that for an L1, O1 machine. Note that the computation *time* for LU scales a factor of *n* faster than the data set. This means that the parallel execution time required for the problem size that achieves the "desirable" parallel efficiency LU grows much more quickly than the table indicates. The time on an L16, O16 machine would be 36 times that on an L1, O1 machine without prefetching and 7.4 times longer with prefetching, even though the data set size required is only 11 times and 3.8 times larger, respectively.

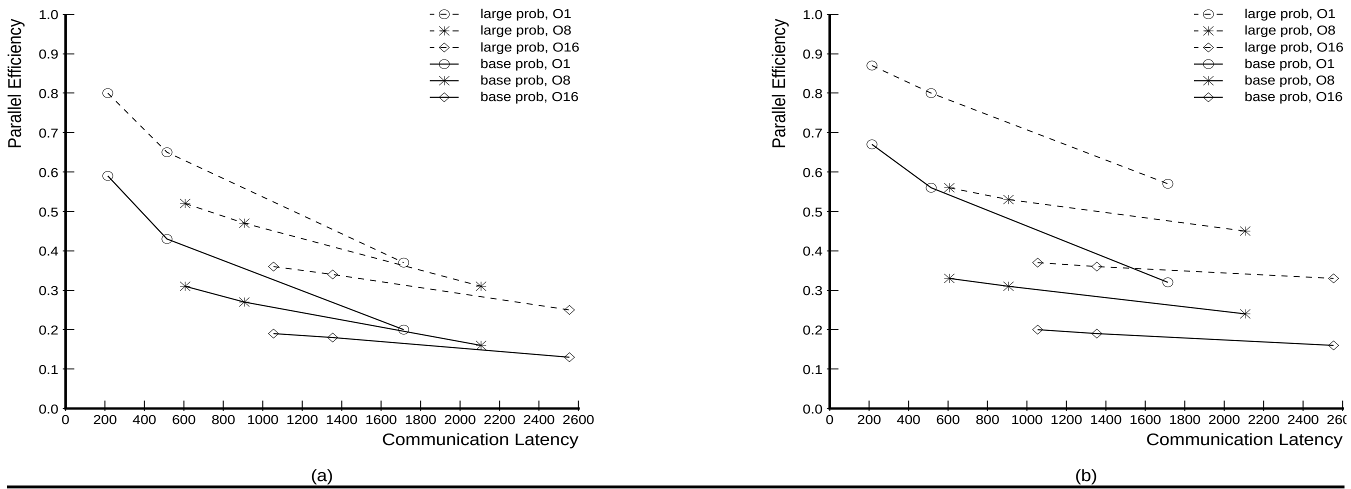


Figure 5.1. Efficiency in Ocean at two different problem sizes for both the (a) non-prefetched and (b) prefetched versions

Ocean: Ocean, which uses nearest-neighbor iterative computations including multigrid, also has a computation-to-communication ratio that scales linearly with problem size, and has a better load balance than LU. As Figure 5.1 shows, both the non-prefetched and prefetched versions of Ocean scale much better than the previous applications. An important observation is that although even the higher occupancy curves increase substantially in efficiency with larger problem sizes, they still do not assume the shape of the lower occupancy curves. Once again, this is because Ocean also has structured communication, so contention does not decrease with increasing problem size. Table 5.4 shows the problem sizes required for 60% efficiency.

Table 5.4. Minimum Problem Size Required for 60% Parallel Efficiency for both Non-Prefetched and Prefetched Ocean

Controller Occupancy	Network Latency					
	Non-Prefetched			Prefetched		
	L1	L4	L16	L1	L4	L16
O1	258^2 (1.0x)	514^2 (4.0x)	1282^2 (25x)	258^2 (1.0x)	386^2 (2.2x)	642^2 (6.2x)
O8	642^2 (6.2x)	898^2 (12x)	1666^2 (42x)	642^2 (6.2x)	642^2 (6.2x)	770^2 (8.9x)
O16	1282^2 (25x)	1410^2 (30x)	2050^2 (63x)	1026^2 (16x)	1026^2 (16x)	1154^2 (20x)

Unlike LU, in Ocean both the data set size and the execution time nominally grow as $O(n^2)$ in the grid dimension. However, the implications of latency and occupancy for execution time are nonetheless more severe than for data set size. This is because increasing data set size also requires scaling other parameters (such as the accuracy used in the multigrid solver and the number of times-steps), which increase execution time further [SHG93]. In fact, the numbers for data set size in Table 5.4 are themselves optimistic, since a larger number of grid points causes more time to be spent in the multigrid equation solver, which has the lowest computation-to-communication ratio and the worst load imbalance in the application.

Finally, the effect of contention on required problem size is much less for Ocean than it is for FFT. For example, for both versions of Ocean the required problem size for the O8 and O16 controllers would have been at most 4 times smaller if they had the same T_C as an O1 controller. Like FFT, the effect of contention is greater in the prefetched version of the code.

Barnes: Unlike the previous applications, Barnes does not have separate phases of communication and computation (though there are more structured versions of the application, written for message passing machines, that do [Salmon90]). As problem size increases, more computation is done between communications, so contention decreases. Since the computation-to-communication ratio also depends on the distribution of particles, predicting the problem size required for 60% efficiency is difficult. However, similar hierarchical N-body applications have an expected computation-to-communication ratio that is linear in the problem size [Katz89]. This suggests that scaling hierarchical N-body applications to retain a desired efficiency should be relatively easy, if communication is the primary bottleneck. Unfortunately, the bottleneck in a cache-coherent shared address space is typically load imbalance, and it is difficult to predict how that improves with problem size since there are different computational phases with different levels of imbalance. Doubling the problem size for Barnes improved performance somewhat at higher occupancies, but not much. However, for the sizes of problems that are run on machines today, we expect that most of the controller configurations we study should perform quite well.

Water: Table 5.5 summarizes the minimum problem sizes required for Water. Water also has a computation-to-communication ratio that scales linearly with data set size. The effect of contention on required problem size is less in Water then it is in all the other applications. In fact, at ATM latencies the O8 controller achieves 60% efficiency at the same problem size whether or not it has its actual value of T_C or it has the T_C of an O1 controller. Contention becomes more important at lower network latencies. For example, the problem size required to achieve 60% efficiency is over 50% larger on an L1, O16 machine than on an L16, O1 machine. Also, note that the execution time for Water grows as the square of the data set size shown in Table 5.5.

Table 5.5. Minimum Problem Size Required for 60% Parallel Efficiency for Water

Controller Occupancy	Network Latency		
	L1	L4	L16
O1	512 (1.0x)	896 (1.8x)	1792 (3.5x)
O8	1152 (2.3x)	1536 (3.0x)	3072 (6.0x)
O16	3072 (6.0x)	3072 (6.0x)	6144 (12.0x)

Overall, substantial increases in problem size are necessary for the lower-performance networks and controllers to achieve the desired efficiency, although the amount of increase varies depending on the specific type of application. There are many important classes of applications (transform methods, sorting) for which the efficiency lost by a less aggressive architecture—in latency or occupancy—is extremely difficult or impossible to regain by increasing problem size. In most of the applications, contention owing to the occupancy of the communication controller played an important role in determining the required growth in problem size, and the contention component of communication was not reduced by increasing problem size. Finally, the effects of increasing latency and occupancy on application execution time, which may be most important, are often more severe than on the growth of its data set size.

6 Conclusions

DSM machines can be characterized in terms of four fundamental parameters: network latency, controller occupancy, node-to-network bandwidth, and the number of processors. Simulating a fixed number of processors (64), and a high bandwidth interconnect, we evaluated the performance impact of network latency and controller occupancy over a range of representative scientific applications. Our results showed: first, that it is possible to achieve good parallel efficiency for a range of applications on machines with low-occupancy, hardwired or special-purpose communication controllers and low-latency MPP networks. Interestingly, the bandwidth requirements for the applications we studied were low in comparison with the node-to-network bandwidth of current MPP networks, indicating that node to network bandwidth is not a major bottleneck.

Our main result, however, is that the occupancy of the communication controller is critical to good performance in DSM machines. For machines with tightly-coupled MPP networks we found that controller occupancy has a large performance impact regardless of whether or not applications incorporated prefetching as a latency hiding technique. For machines with loosely-coupled networks, we showed that without latency hiding, controller occupancy did not matter to overall performance. But, with latency hiding, controller occupancy once again became a performance bottleneck. Since machines with high-latency networks will need to incorporate latency hiding whenever possible to obtain good performance, these results show that it is important to use low-occupancy communication controllers at any network latency.

Moreover, it was not just the latency component of the higher occupancy controllers that caused performance degradation, but rather the contention component, even without latency hiding. This contention component proved difficult to model analytically, but we found that several important classes of applications communicate in “bulk synchronous” phases where the computation-to-communication ratio is constant, and therefore the contention related effects do not decrease with increasing problem size. Using this insight, our results showed that for many classes of applications, it is extremely difficult for architectures with higher values of network latency or controller occupancy to achieve high parallel efficiency by scaling up the problem size. That is, the problem size needed to maintain the desired efficiency quickly becomes unreasonable. There are applications that attain the desired efficiency with reasonable data set sizes, although for many of these applications the parallel execution time scales much faster than the required data set size. This does not bode well for current and proposed architectures that provide communication and coherence with cache-line granularity with general-purpose processors on the memory bus or I/O bus of the main processor, or for architectures with high-latency networks. Fortunately, the controller occupancies of specialized or hardwired controllers on the memory bus were low enough to achieve the desired performance goal for all the applications in this study.

The tendency among DSM designers has been to focus on network latency and network bandwidth as the important performance issues in the communication architecture. Our results demonstrate that the occupancy of the communication controller is just as important to overall performance, if not more so.

Acknowledgments

This work was supported by ARPA contract DABT63-94-C-0054. We would like to thank Edward Rothberg for his help with the methodology of this paper. We would also like to thank Steven Woo for his work on the SPLASH-2 application suite, and Anoop Gupta, Jeffrey Kuskin, and David Ofelt for their comments on early drafts of this paper.

References

[ABC+95] Anant Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2-13, Santa Margherita Liguere, Italy, June 1995.

[BLA+94] M. Blumrich et al. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142-153, April 1994.

[CKP+93] David Culler et al. LogP: Toward a realistic model of parallel computation. In *Proceedings of the Principles and Practice of Parallel Processing*, pages 1-12, 1993.

[GMB88] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for a 1024-processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9 No. 4, pages 609-638, 1988.

[Golds93] Stephen Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[HKO+94] Mark Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274-285, San Jose, CA, October 1994.

[HS94] Chris Holt and Jaswinder Pal Singh. Hierarchical N-Body Methods on Shared Address Space Multiprocessors. *SIAM Conference on Parallel Processing for Scientific Computing*, February 1995, to appear.

[Katz89] Jacob Katzenelson. Computational Structure of the N-body Problem. *SIAM Journal of Scientific and Statistical Computing*, pages 787-815, July 1989.

[KOH+94] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, April 1994.

[KSR92] Kendall Square Research. KSR1 Technical Summary. Waltham, MA, 1992.

[LC96] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308-317, Philadelphia, PA, May 1996.

[LLA+97] James Laudon et al. System Overview of the SGI Origin 200/2000 Product Line. To appear in COMPCON 1997.

[LLG+92] Daniel Lenoski et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63-79, March 1992.

[NAB+95] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. ICPP 1995.

[Roth93] Edward Rothberg. Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization. Ph.D. Thesis, Stanford University, January 1993.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 34-43, Philadelphia, PA, May 1996.

[RSG93] Edward Rothberg, Jaswinder Pal Singh and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14-25, San Diego, CA, 1993.

[Salmon90] John K. Salmon. Parallel Hierarchical N-body Methods. Ph.D. Thesis, California Institute of Technology, December 1990.

[SFL+94] Ioannis Schoinas et al. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297-306, San Jose, CA, October 1994.

[SHG93] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: methodology and examples. *IEEE Computer*, July 1993.

[SSA+95] Craig B. Stunkel et al. The SP2 High-Performance Switch. *IBM Systems Journal*, vol. 34, no. 2, 1995.

[WH95] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, February 1995.

[WOT+95] Steven Cameron Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24-36, Santa Margherita Liguere, Italy, June 1995.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219-229, San Jose, CA, October 1994.