Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation

Mainak Chaudhuri, *Student Member*, *IEEE*, Mark Heinrich, *Member*, *IEEE*, Chris Holt, Jaswinder Pal Singh, *Member*, *IEEE*, Edward Rothberg, and John Hennessy, *Fellow*, *IEEE*

Abstract—While the desire to use commodity parts in the communication architecture of a DSM multiprocessor offers advantages in cost and design time, the impact on application performance is unclear. We study this performance impact through detailed simulation, analytical modeling, and experiments on a flexible DSM prototype, using a range of parallel applications. We adapt the logP model to characterize the communication architectures of DSM machines. The l (network latency) and o (controller occupancy) parameters are the keys to performance in these machines, with the g (node-to-network bandwidth) parameter becoming important only for the fastest controllers. We show that, of all the logP parameters, controller occupancy has the greatest impact on application performance. Of the two contributions of occupancy to performance degradation—the latency it adds and the contention it induces—it is the contention component that governs performance regardless of network latency, showing a quadratic dependence on o. As expected, techniques to reduce the impact of latency make controller occupancy a greater bottleneck. Surprisingly, 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. Through experiments on a DSM prototype, we show that there are important classes of applications for which the performance lost by using higher occupancy controllers cannot be regained easily, if at all, by scaling the problem size.

Index Terms—Occupancy, distributed shared memory multiprocessors, communication controller, latency, bandwidth, queuing model, flexible node controller.

1 Introduction

Distributed shared memory (DSM) multiprocessors are converging to a family of architectures that resemble a generic system architecture. This architecture consists of a number of processing nodes connected by a general interconnection network. Every node contains a processor, its cache subsystem, and a portion of the total main memory on the machine. It also contains a communication controller that is responsible for managing the communication both within and between nodes. Our interest in this paper is in the specific class of cache-coherent DSM machines.

There are many ways to build cache-coherent DSM machines, arising from differences in desired performance and cost characteristics and in the extent to which one wants to use commodity parts and interfaces rather than build 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.

DSM networks vary in their latency and bandwidth characteristics, as well as in their topologies. They range from low-latency, high-bandwidth MPP networks, all the way to commodity local area networks (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 the differences in design cost and design effort, all of these architectures are viable. Current and proposed architectures for cache-coherent DSM machines take different positions on the above trade offs and, thus, there are examples of real machines at almost every point in this design space. The question we address in this paper is how the performance characteristics of the network and controller affect how well the machines will run parallel programs written for cache-coherent multiprocessors. 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 wide range of computations. We address this question by studying a range of important computations and communication architectures through a combination of detailed simulation, analytical modeling, and experiments on a flexible DSM prototype.

- *M. Chaudhuri and M. Heinrich are with the Computer Systems Laboratory, Cornell University, Ithaca, NY 14853. E-mail: {mainak, heinrich}@csl.cornell.edu.*
- *C. Holt is with Transmeta, Inc., 3940 Freedom Circle, Santa Clara, CA 95054. E-mail: xris@transmeta.com.*
- *J.P. Singh is with the Department of Computer Science, Princeton University, Princeton, NJ 08544. E-mail: jps@cs.princeton.edu.*
- *E. Rothberg is with ILOG, Inc., 1901 Landings Dr., Mountain View, CA 94043. E-mail: rothberg@ilog.com.*
- *J. Hennessy is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305. E-mail: jlh@mojave.stanford.edu.*

Manuscript received 23 Jan. 2002; revised 12 Aug. 2002; accepted 27 Aug. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115758.

We characterize the communication architectures of DSM multiprocessors by a few key parameters that are similar to those in the logP model [6]. Our characterizations and the design space that they represent are described in Section 2. Section 3 describes the framework and methodology we use to study the effectiveness of different types of DSM architectures. Section 4 presents and analyzes our simulation results. Section 5 presents a queuing model to analyze the contention in the communication controller and uses that model to predict the parallel efficiency of applications running on different communication architectures. Section 6 describes the effect of varying the occupancy of a programmable protocol engine in a flexible DSM architecture and shows that it is very difficult to regain the lost performance by increasing the problem size as the controller becomes slower. Section 7 concludes the paper.

1.1 Related Work

The logP model suggested in [6], introduced a machine-independent model to reason about the performance of message-passing parallel programs. In a 1995 technical report [13], we first adapted this model to describe a generic DSM architecture, where o was the occupancy of the DSM communication controller, and carried out a simulation-based study to show the effects of latency (l) and occupancy (o) on the performance of large-scale parallel applications and computational kernels. This was followed by a similar study by others on a high-performance NOW [18]. The effects of processor overhead, network interface occupancy, node-to-network bandwidth, and interrupt overhead have also been studied in the context of shared virtual memory clusters [2].

Inspired by our previous study, many research groups have proposed designing controllers with lower occupancy [20] and have explored methods to reduce the contention of the communication controller [7], [9], [12], [19], [21], [34]. Also, it has been suggested that, if the controller is slower than the node-to-network interface, increasing the coherence granularity may help reduce contention [33]. The effects of bisection bandwidth and ratio of processor cycle time to network latency have been studied for several versions of shared memory and message passing applications running on the Alewife machine [1]. However, this study does not discuss the performance effects of node controller occupancy and node-to-network bandwidth, or how these parameters interact with each other as one moves from one point to another in the design space. In [27], a performance model for shared memory machines is presented as a function of various architectural and statistical parameters of the system. We present a much more simple analytical model in this paper and show how the model behaves with varying communication architecture parameters. In this paper, we expand the ideas in our original report, make the analysis more concrete with a queuing model, and augment the simulation results with experimental results obtained from a programmable DSM prototype. The experimental results allow us to look at the effects of controller occupancy at larger problem sizes than it is possible to simulate and determine whether less aggressive communication controllers can regain their lost performance at these larger problem sizes.

2 Parameters and Design Space

Using the logP model, we abstract the multiprocessor 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 flit of a message enters the network at a source node to the moment 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 through the network interface), and P is the number of processors. The only difference between our DSM model and the logP model developed for message-passing machines 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 or service rate. However, since controller bandwidth may be confused with (the very different) network bandwidth parameter, we prefer to use the term controller occupancy.

Our original study fixed the number of processors at 64. In this paper, we simulate two values of P, $P = 32$ and $P = 64$, and we carry out a study on the effect of varying occupancy on a real 16 and 32-node DSM multiprocessor with a programmable protocol engine. We also briefly explore the effect of speeding up the main processor relative to the memory system. 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 in the context of realistic machines.

2.1 Latency

The latency of a message through the network depends on, among other things, how many hops the message travels in the network. For the moderate-scale machines that we consider ($\leq$ 64 processors), the overhead of getting the message from the processor into the network and vice versa usually 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 a two-dimensional mesh topology. By taking into account the topology-related effects, our experimental model can be easily adapted to the cases where latency is not homogeneous over the entire network or when there are some nonnegligible variations of latency over time. However, this study is beyond the scope of this paper.

2.2 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, hence, incur occupancy. 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 the individual transactions may not be the same. While we would like to represent occupancy by a single value of o, occupancy in real machines often depends on the type of the transaction. Let us examine these issues separately.

Clearly, all events related to internode communication and protocol processing incur controller occupancy. These 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 [25]. However, note that we do take into account the contention between the main processor and the communication controller in accessing local memory. We also assume that the state lookup that determines if a local 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 [11]. In this case, we assume the data 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 $2o + C$. Occupancy is also incurred 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 one system clock cycle per invalidation that it sends. In addition, occupancy is incurred while receiving acknowledgments corresponding to certain requests (e.g., invalidation acknowledgments) and while receiving ownership transfer messages (e.g., sharing writebacks). The controller handles these messages similarly to normal replies, incurring an occupancy of o.

2.3 Bandwidth or Gap

The gap (g) parameter specifies the reciprocal of node-to-network bandwidth. It determines how fast data can be transferred through the network interface (between the communication controller and the network itself). Our original study did not vary node-to-network bandwidth. In this paper, we explore the effect of varying g over a wide range of values. While studying the effects of l and o only, we fix $1/g$ at 400 MB/s peak, which corresponds to MPP networks on recent machines. For coherence messages that do not carry data, the occupancy of the communication controller always dominates this gap limitation. For messages that carry data, the gap parameter can theoretically become the bottleneck before controller occupancy for the two lowest occupancies we examine. We show that this is actually the case for some applications.

2.4 Design Space

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 processor interface and network interface delays into and out of each controller (PI_{in}, PI_{out}, NI_{in}, and NI_{out}), leads to a total round-trip latency as seen by the processor (without any contention) of $PI_{in} + o + NI_{out} + l + NI_{in} + 2o + NI_{out} + l + NI_{in} + o + PI_{out}$ for the miss, or $2l + 4o + PI_{in} + PI_{out} + 2(NI_{in} + NI_{out})$. 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 requester or the home), the latency would be $3l + 6o + C + PI_{in} + PI_{out} + 3(NI_{in} + NI_{out})$. However, this is only the latency seen by the requester. The controller at the home node of the request has to handle a subsequent ownership transfer reply. The total latency of this transaction is given by NI_{out} at the previous owner, plus l to traverse the network, plus $NI_{in} + o$ at the home, leading to a total latency of $l + o + NI_{in} + NI_{out}$.

The network latency l and the controller occupancy o are the variables in the above costs. In the analysis presented above, we assume that data transmission/reception through network interfaces is completely pipelined and is completely overlapped with other activities in the communication architecture. Therefore, we do not include g-related latency terms in this analysis. Additive g-related latency terms may appear in systems with fast controllers having very slow network interfaces. But, we will show that this is most often not the case in practice.

We focus on a range of values for l and o, as shown in Tables 1 and 2, covering a variety of possible architectural alternatives. Our latencies (l) vary from tightly coupled,

TABLE 1
Network Latencies in the Design Space

Arch. Parameter	Hardware Description	Average Latency (System Cycles)
L_1	Aggressive MPP	25
L_2, L_4, L_8	Distributed MPP	50, 100, 200
L_{16}, L_{32}	Commodity LAN	400, 800

low-latency MPP networks, through physically distributed MPP networks, all the way to LANs composed of commodity switches. Table 1 shows the average latency for 64 processors. Our system cycles correspond to a 100 MHz system clock frequency. Table 2 describes the controller occupancies in our design space. Small values of occupancy represent communication controllers that are tightly integrated, hardwired state machines. Such controllers appear in the MIT Alewife machine [1], the KSR1 machine [15], the Stanford DASH multiprocessor [17], and the SGI Origin 2000 [16]. As o increases, the controller becomes less hardwired and more general-purpose, from specialized coprocessors like those in the Stanford FLASH multiprocessor [14] and the Sun S3.mp [22], through inexpensive off-the-shelf processors on the memory bus as in Typhoon-1 [23], to a controller on the I/O bus of the main processor like those in SHRIMP [3], and the IBM SP2 [28]. We also vary the node-to-network bandwidth from 400 MB/s (g_1) down to 25 MB/s (g_{16}), to analyze the effect of reducing network bandwidth on the applications under consideration.

3 Framework and Methodology

The applications [31] and the base problems sizes that we use in our simulation study are summarized in Table 3. They include three complete applications (Barnes-Hut, Ocean, and Water) and three computational kernels (FFT, LU, and Radix-Sort). The programs were chosen because they represent a variety of important scientific computations with different communication patterns and requirements. Descriptions of the applications can be found in: Barnes-Hut [26], Radix-Sort and Ocean [32], Water [31], and FFT and LU [24]. The communication characteristics of the applications can be found in [24], [31]. The applications are highly optimized to improve communication performance, particularly to reduce spurious hot-spotting or contention effects that adversely impact controller occupancy. Even with these optimizations, we will show that occupancy still remains an important determinant of performance. The codes for the applications are taken from the SPLASH-2 application suite [31], although Radix-Sort was modified to use a tree data structure (rather than a linear key chain) to communicate ranks and densities efficiently.

We explore the performance effects of varying l, o, g, P and the problem sizes of these applications. The standard definition of parallel efficiency is used as the metric to measure the performance of a particular communication architecture or a particular problem size. Parallel efficiency is defined as the speedup over a sequential implementation of the application on a uniprocessor, divided by the number of processors (P). Some machine designers argue that cost-performance is the best overall figure of merit [30]. 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

TABLE 2
Controller Occupancies in the Design Space

Arch. Parameter	Hardware Description	Occupancy (System Cycles)
O_1	Hardwired	7
O_2, O_4	Customized Co-proc.	14, 28
O_8	General-purpose Co-proc. on memory bus	56
O_{16}	General-purpose Co-proc. on I/O bus	112

TABLE 3
Applications, Communication Patterns, and Base Problem Sizes

Application	Description	Communication Pattern	Problem Size
Barnes-Hut	Barnes-Hut hierarchical N-body simulation	irregular, hierarchical	8192 particles
Ocean	Multigrid large-scale ocean simulation	nearest neighbor iterative	514×514 grid
Water	Molecular dynamics simulation	structured, many-to-many	1024 molecules, 3 time steps
FFT	Radix-$\sqrt{n}$ six-step Fast Fourier Transform	all-to-all, blocked	1M points
LU	Blocked dense LU decomposition	structured, one-to-many	512×512 matrix, 16×16 block
Radix-Sort	Integer radix sort	irregular, all-to-all	2M keys, radix 256

use a pure performance metric and keep the study free of cost issues. If designers want to spend less money and use cheaper, slower components, 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.

In this paper, we present simulation results as well as experimental results gathered from an existing programmable DSM prototype. The simulator models contention in detail within the communication controller, between the controller and its external interfaces, at main memory, and for the system bus. The input and output queue sizes in the controller's processor interface are uniformly set at 16 and two entries, respectively, while those for the network interface are uniformly set at two and 16 entries, respectively. We assume processor interface delays of one system cycle inbound and four system cycles outbound and network interface delays of eight system cycles inbound and four system cycles outbound. We assume that the latencies through the interfaces remain fixed as controller and network characteristics are varied. We also fix the access time of main memory DRAM at 140 ns (14 system cycles), resulting in a local read miss time of 190 ns, one system cycle faster than the SGI Origin 2000. Fixing the interface delays and the memory access time is realistic [11] and allows us to focus on the performance of the communication architecture and the effects of varying l, o, g and P.

The processor controls its own secondary cache, and the simulator uses 27 processor cycles (5 ns each cycle) for C, the time it charges the controller to retrieve state information from the processor cache when necessary. This latency is close to the latencies reported in previous studies [11]. There are separate 64 KB primary instruction and data caches that are two-way set associative and have a line size of 64 bytes. The secondary cache is unified, 2 MB in size, two-way set associative, and has a line size of 128 bytes. We also assume that the processor ISA includes a prefetch instruction. In our processor model, a load miss stalls the processor until the first double-word of data is returned, while prefetch and store misses will not stall the processor unless there are already references outstanding to four different cache lines.

4 Simulation Results

This section presents and analyzes the simulation results of all the six SPLASH-2 applications that we are looking at.

4.1 What We Expect To See

As l and o increase for fixed values of g and P with a given problem size, we expect that parallel efficiency should decrease. To get a rough idea about *how* the parallel efficiency should vary with l and o, we use the model of parallel efficiency we suggested in [13]:

$$\eta = \frac{T_{comp}}{T_{comp} + V_{comm}(T_L + T_C)}, \quad (1)$$

where T_{comp} is the uniprocessor computation time, V_{comm} is the total volume of communication, and T_L and T_C are the average stall times due to latency and contention, respectively, for each communication. We define communication to be any transaction that incurs occupancy on the communication controller. Note that T_L includes the latencies for all protocol transactions, not just remote read misses clean at the home. Equation 1 is considered here, only to get some intuitive idea about the expected results. The readers should not take it as a formal definition of parallel efficiency, although this equation models the parallel efficiency fairly well under the assumptions of perfect load-balance and an equal distribution of volume of communication across the nodes in the system. But, this equation fails to explain the well-known phenomenon of superlinear speedup that may happen due to cache effects related to the problem size on one versus multiple processors. The parallel efficiency of our simulation runs is calculated as speedup divided by the number of processors. We do not use (1) for that purpose.

For a fixed problem size, a fixed number of processors, and fixed g, both T_{comp} and V_{comm} are constants. We will show that T_L varies linearly with l and o. To see why this is true, observe that the uncontended latency of any transaction is given by $al + bo + c$, where a and b are constants that depend on the type of the transaction and c is a constant that depends on the time spent in various interfaces between the communication controller, the processor, and the network. The average over all these uncontended latencies will have the same linear behavior. Finally, we turn to T_C, the average contention in the communication controller. If the contention in the controller was fixed at a constant value as we traverse the design space, we would see the same parallel efficiency for various values of o as long as we hold T_L at a constant value. On the other hand, if T_C increases with increasing o, we would expect to see a gradual decrease in parallel efficiency as we move from O_1 to O_{16} for a fixed value of T_L.

Next, we explore the question of varying the gap (i.e., the node-to-network bandwidth). As node-to-network bandwidth decreases, we expect to see a decrease in parallel efficiency. The importance of g depends on the node-to-network bandwidth requirement of the application under consideration. Although the average bandwidth requirements of the applications reported in [13] were less than the capacity of the network, we will see that g can still be important if the communication pattern is bursty.

The next two dimensions in our design space are P and the problem size. With increasing P, we expect to see a decrease in parallel efficiency for the same communication architecture. This is simply because T_C increases as P increases. Further, the average number of hops a message needs to travel also increases leading to a corresponding increase in T_L. We also expect that increasing problem size, up to a point, will increase parallel efficiency. But, the effect of problem size depends on the communication-to-computation ratio of the application, and the question that remains is: How big does the problem size need to be for less aggressive architectures to regain their lost performance, if it is possible at all?

In the following simulation results, we will focus on both prefetched and nonprefetched applications. Since prefetching can introduce extra and, sometimes, unnecessary communication traffic (if prefetching is not timely), in Table 4, we show how effective prefetching was for FFT, Ocean, Radix-Sort, and LU in an $L_1O_1g_1$ simulation for the base problem sizes and 64 processors. Prefetching is effective if the read miss rate is reduced without increasing

TABLE 4
Effect of Hand-Inserted Prefetches

Application	L1 Data Cache Read Miss Rate	L1 Data Cache Write Miss Rate	Number of Prefetches	L1 Data Cache Prefetch Miss Count
Non-pref. FFT	0.55%	2.10%	N/A	N/A
Pref. FFT	0.33%	2.06%	774144	774144
Non-pref. Ocean	1.97%	19.01%	N/A	N/A
Pref. Ocean	1.82%	18.93%	731192	730856
Non-pref. Radix	2.18%	7.43%	N/A	N/A
Pref. Radix	1.83%	7.51%	140072	138580
Non-pref. LU	0.33%	7.26%	N/A	N/A
Pref. LU	0.33%	7.33%	37492	23488

the write miss rate. For FFT, prefetching was found to be quite effective because it reduced the L1 data cache read miss rate from 0.55 percent to 0.33 percent without significantly changing the L1 data cache write miss rate (in this case, write miss rate also decreased from 2.1 percent to 2.06 percent). Also, all the prefetches missed in the L1 data cache, meaning that all of the prefetch instructions were useful. For Ocean and Radix-Sort, prefetching was effective for the L_1 network latency, but we found that it could not hide the latency well as the network approached less aggressive MPP networks and commodity LANs. Finally, for LU, prefetching did not help in reducing the read miss rate. This is mostly because of a very small number of prefetch misses (23,488) compared to the total number of load misses (1,008,479) in the L1 data cache, which, in turn, is due to the fact that the total number of prefetches is small compared to the total number of loads. Also, prefetched LU may introduce certain hot-spots in the memory system because, during the perimeter update phase, all the processors owning the perimeter blocks may try to send prefetches to the owner of the corresponding diagonal block at the same time. For all the applications, almost all prefetches missed in the L1 data cache. Therefore, prefetching did not introduce any unnecessary instruction overhead.

4.2 Case Studies: FFT and Ocean

First, we select two representative applications from the SPLASH-2 application suite to explore, in detail, how l, o, g, P, and the problem size affect the performance of DSM multiprocessors. We select FFT because it is an easily understood application that has a regular communication pattern, and we select Ocean because it is a complex, large-scale application.

4.2.1 Experience with FFT

First, we examine the effects of l and o on nonprefetched and prefetched FFT. In our simulations, the ratio of processor clock speed to the system clock speed is set to two. Increasing this ratio is equivalent to increasing the processor clock rate or, alternatively, to having a more aggressive superscalar processor that can issue requests to the memory subsystem at a faster rate [10]. We will vary this ratio as a part of our case study, and we will see that higher ratios will result in worse parallel performance due to a higher T_C and a smaller T_{comp} for the same problem size. Fig. 1 plots parallel efficiency against average communication latency (T_L) in processor clock cycles for nonprefetched and prefetched FFT with the base problem size (1M points) running on 64 processors. Different curves for different values of o indicate that we do have occupancy-induced contention in the node-controller. The six points along each o-curve corresponds to the six network architectures ranging from L_1 (tightly coupled MPP latency) to L_{32} (commodity LAN latency). In this paper, all the efficiency curves that show effects of only l and o have a constant node-to-network bandwidth of 400 MB/s (a g_1 configuration).

Without prefetching: As already indicated, the multiple efficiency curves show that the contention component of the

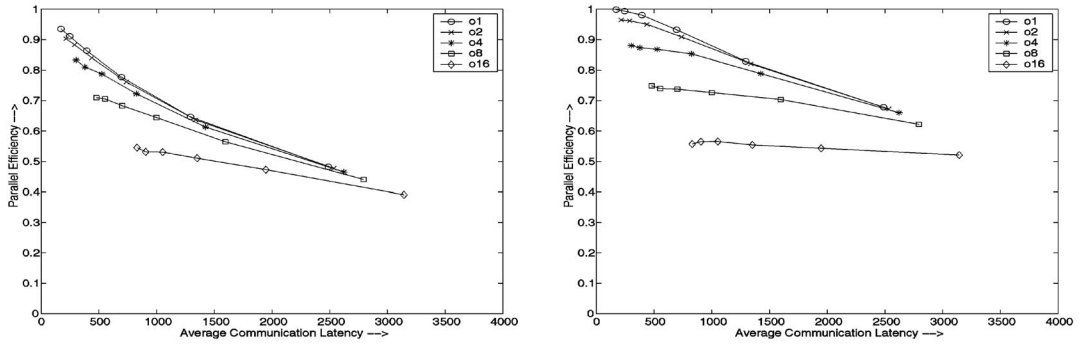


Fig. 1. (a) Nonprefetched and (b) prefetched 1M-point FFT running on 64 processors.

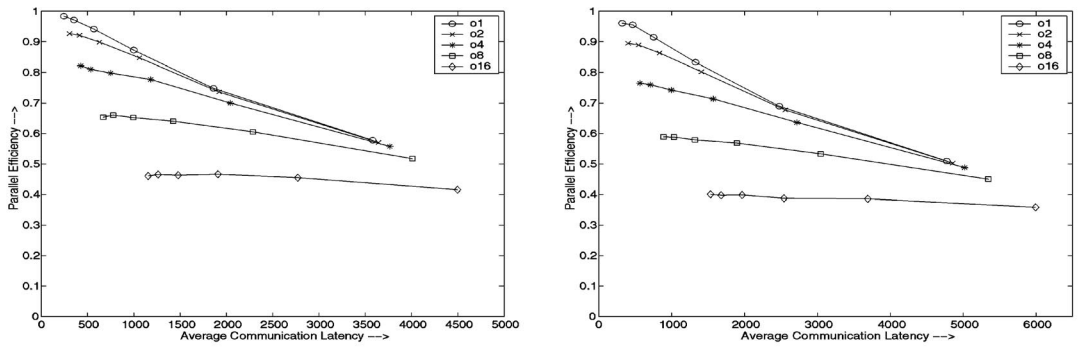


Fig. 2. Prefetched 1M-point FFT with processor/memory speed ratios of (a) 3 and (b) 4 on 64 processors.

controller is indeed important, even without prefetching. 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, regardless of the network latency.

Note that all efficiency curves nearly converge at high values of l, implying that, at today's commodity network 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. Increases in T_C account for the efficiency lost while communication latency is held constant and controller occupancy is increased.

With prefetching: In this case, there are also multiple parallel efficiency curves that flatten out as o increases. Unlike the nonprefetched case, the curves no longer converge at commodity LAN latency because the contention component of occupancy affects overall performance, even at high network latencies. At our highest network latency, an O_1 machine is 1.5 times faster than an O_{16} machine in the prefetched case, but only 1.3 times faster in the nonprefetched case. Prefetching improves performance more at low o and low-to-moderate l than it does at higher values of o and l. At moderate l, prefetching cannot hide all the network latency and increases in latency begin to hurt the prefetched case at the same rate as the nonprefetched case. At medium o, the controller becomes a bottleneck, as it is unable to match the increased bandwidth needs of prefetching. We see, therefore, to support prefetching in DSM machines, it is crucial to keep controller occupancy low.

Effect of faster processors: The performance gap between the processor and the memory subsystem is ever-increasing. Fig. 2 shows the efficiency curves for processor/memory speed ratios of 3 and 4. As expected, the shapes of the curves remain unchanged, while the parallel efficiency correspondingly decreases. Also, note that the decrease in parallel efficiency is more for slower controllers than the low-occupancy ones. For example, with an agressive MPP network (L_1 configuration), the parallel efficiency for an O_1 controller drops from 0.98 to 0.96 as the system moves from a speed ratio of 3 to 4. On the other hand, for an O_{16} controller with the same network, the parallel efficiency drops from 0.47 to 0.39 as the speed ratio changes from 3 to 4. This drop is significant given that an efficiency of 0.47 corresponds to a speedup of 30.08, while 0.39 corresponds to a speedup of 24.96 on a 64-processor system, which in turn translates into a large difference in execution times. This suggests that, as the gap between the clock rates of the processor and the memory subsystem continue to increase, DSM controllers will need to become more tightly integrated and have even lower occupancy. The remainder of our simulation results uses our base processor/memory speed ratio of two. This is generous toward less aggressive controllers and networks, yet, even so, we will see that their performance in DSM systems is still poor.

Effect of varying node-to-network bandwidth: Next, we explore the effect of varying g on FFT. Fig. 3 plots the

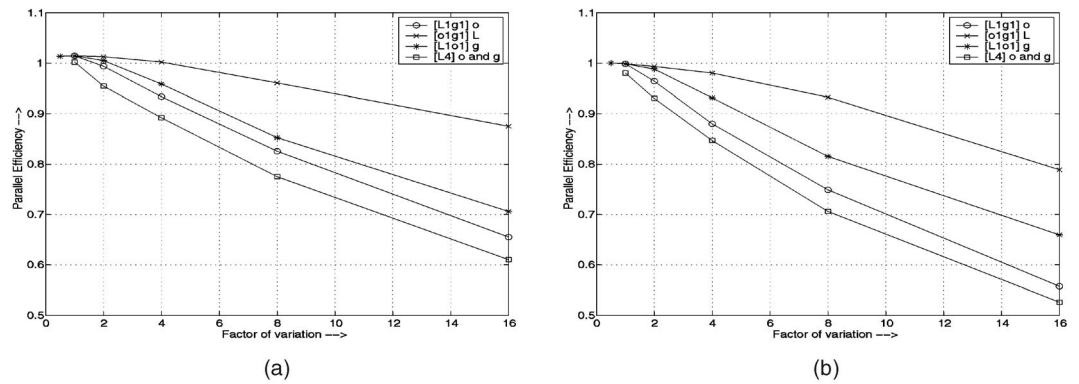


Fig. 3. Effect of varying l, o, and g on prefetched 1M-point FFT for (a) 32 and (b) 64 processors.

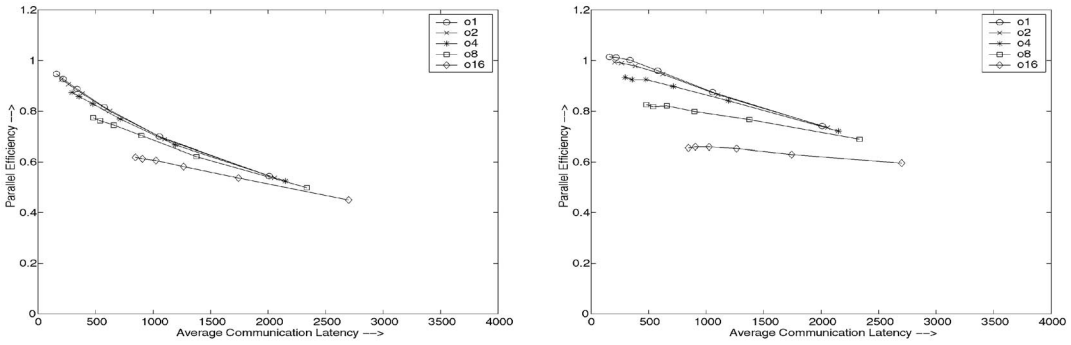


Fig. 4. (a) Nonprefetched and (b) prefetched 1M-point FFT running on 32 processors.

parallel efficiency of FFT on 1M points for both 32 and 64 processors. The "[L1g1] o" curve exhibits the effect of varying o as we keep the network and the node-to-network interface at the fastest possible level (L_1g_1). The x-axis plots the factor of variation from 1 to 16. Similar effects of L are shown in the "[o1g1] L" curve. The effect of slowing down the node-to-network interface is plotted in the "[L1o1] g" curve. The point corresponding to $x = 0.5$ is also plotted, signifying the parallel efficiency when the node-to-network bandwidth is $1/g = 800$ MB/s. Note that we do not lose any efficiency when the bandwidth decreases from 800 MB/s to 400 MB/s. Also, the "[L4] o and g" curve plots the effect of varying o and g together for an L_4 network (distributed MPP). This curve is more relevant to variation in g because the controller speed and the interface bandwidth normally go hand in hand, given that it only makes sense to build a controller with a good balance between controller bandwidth and interface bandwidth. These curves clearly bring out the fact that starting from an $L_1O_1g_1$ configuration, one loses most in terms of performance if controller occupancy is increased. In addition, we see that network latency is the least important parameter for FFT. In fact, for FFT, the order of these three architectural parameters in terms of performance sensitivity is o, then g, then l.

Effect of varying P and the problem size: Fig. 4 shows the parallel efficiency curves just like Fig. 1, but now with 32 instead of 64 processors. As expected, the parallel efficiency for 32 processors is only slightly higher (at most 5 percent for various values of l) than that for 64 processors for O_1, O_2, and O_4 controllers. However, for O_8 and O_{16} controllers, there is a significant gain in efficiency as the number of processors drops to 32. For example, with an $L_{32}O_{16}$ configuration on a 64 processor system, prefetched FFT achieves an efficiency of around 0.5, while a 32 processor system has an efficiency of around 0.6. This is expected because for slow controllers, the effect of contention (which increases with increasing processor-count) is larger as compared to relatively fast controllers. Finally, we explore the effect of varying the problem size in FFT. Fig. 5 shows the parallel efficiency curves for prefetched FFT with a smaller data size (256K points) running on 64 processors. A comparison with Fig. 1 reveals that we do gain in terms of efficiency by increasing the data size from 256K points to 1M points. But, how much do we need to increase the problem size for less aggressive controllers? We cannot simulate larger problem sizes for FFT, but we will shed some light on this question by using a flexible DSM prototype in Section 6.

4.2.2 Ocean

Fig. 6 plots parallel efficiency against average communication latency (T_L) for nonprefetched and prefetched Ocean with the base problem size (514×514 grid). Ocean performs many iterative nearest-neighbor computations on regular grids and depends strongly on network latency. However, its performance is also dependent on controller occupancy, especially for the prefetched version and for low-latency networks (Aggressive MPP). The main problem with Ocean is that it cannot fully exploit the spatial locality of remote data and, hence, it is highly sensitive to network latency, even in the prefetched version.

The effect of increasing the processor/memory speed ratio for Ocean does not have a significant performance impact and we do not show those results here. This supports our claim that Ocean is much less sensitive to controller occupancy than FFT. The effect of varying g is exhibited in Fig. 7. The important observation is that, in an L_1O_1 system with node-to-network bandwidth less than or equal to 50 MB/s, g becomes significantly more important than o in an L_1g_1 system. However, since a tightly integrated controller (L_1O_1) is not likely to have such a poor node-to-network interface, the curve that shows simultaneous variation in o and g gives a much more realistic estimate of the performance impact as these parameters vary. This curve clearly demonstrates the fact that the combined effect of increasing o and g is much more devastating than only increasing l. The absolute value of the

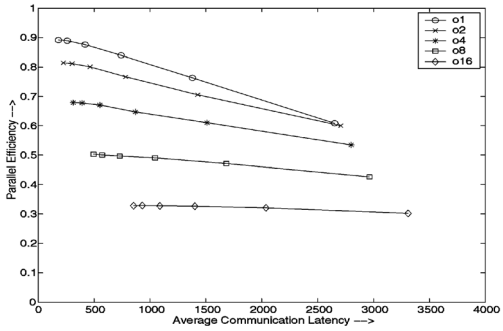


Fig. 5. Prefetched FFT on 256K points and 64 processors.

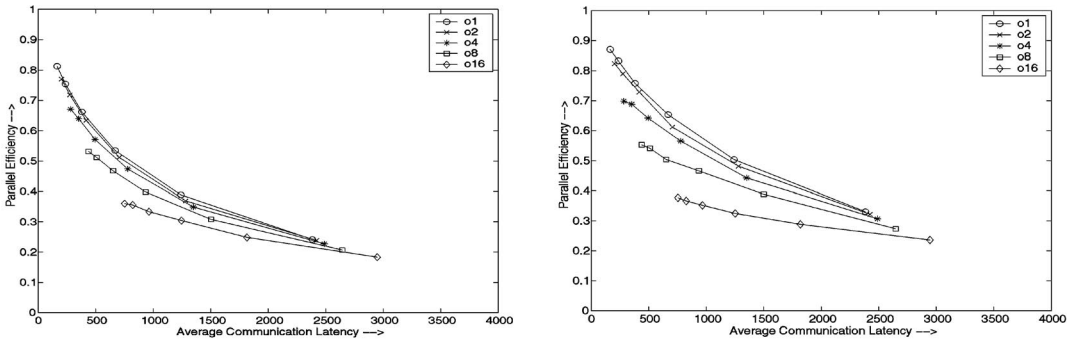


Fig. 6. (a) Nonprefetched and (b) prefetched Ocean running on 64 processors and a 514×514 grid.

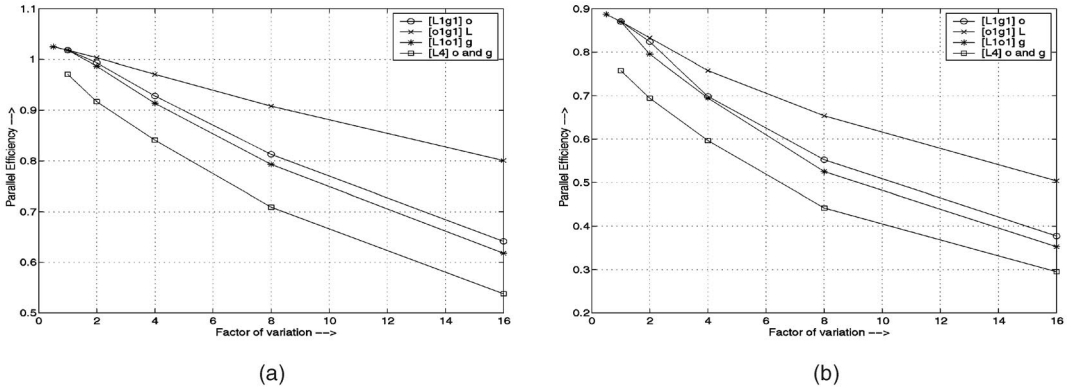


Fig. 7. Effect of varying l, o, and g on prefetched Ocean for (a) 32 and (b) 64 processors with a 514×514 grid.

slope of the latency curve is consistently less than the other three curves. This also supports the view that latency is less important than occupancy and bandwidth. The slopes of the other three curves are similar, although, at some points, the g-curve has a lower slope compared to the o-curve. This means that, for Ocean, the message bandwidth of the controller should be well-balanced with the link bandwidth of the interface; otherwise, one of them will be underutilized and the other one will become the bottleneck.

As we decrease the number of processors from 64 to 32, we observe the same trend as in FFT—parallel efficiency increases. In fact, an L_1O_1 controller achieves superlinear speedup for prefetched Ocean. However, when we change the grid size to 258×258, we observe a big change in performance (see Fig. 8). With the reduced data set, the efficiency achieved by an L_1O_1 controller is less than that of an L_1O_8 controller on the bigger grid size. Again, we will see the effect of occupancy variation with larger problem sizes in Section 6.

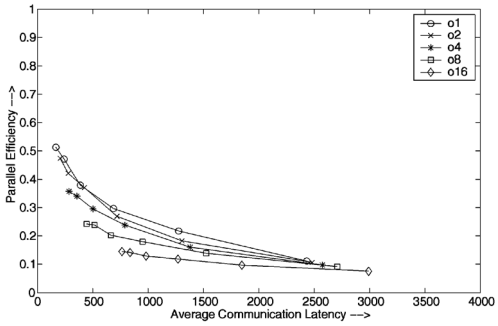


Fig. 8. Prefetched Ocean on a 258×258 grid and 64 processors.

4.3 Other Simulation Results

In the following, we present the remaining simulation results for Radix-Sort, LU, Barnes-Hut, and Water. Since we continue to see similar trends when P is decreased from 64 to 32, we mainly focus on results for 64 processors and point out the effects of varying l, o, and g.

Radix-Sort: The results for Radix-Sort shown in Fig. 9 are similar to FFT, with a few notable exceptions. Like FFT, without prefetching, all the efficiency curves almost converge by today's LAN latencies (our rightmost points). While the O_1 and O_2 controllers have similar performance, the O_8 curve is much flatter than it is in FFT, and the O_{16} curve is almost totally flat. This indicates that, in Radix-Sort, contention induced by slower controllers matters even more than it does in FFT.

In the prefetched version of Radix-Sort, we see a bigger linear dependence on network latency than that in prefetched FFT (i.e., prefetching is not as successful in Radix-Sort as it is in FFT for networks slower than the L_1 configuration because of the irregular sender-initiated bursty communication in the permutation phase). Prefetching helps much more at lower values of o, indicating that it is critical to keep occupancy low when prefetching, even with LAN network latencies.

Fig. 10 shows the effect of varying g. For an L_1O_1 controller, g is much more important than o is for an L_1g_1 controller. This is because of the permutation phase in

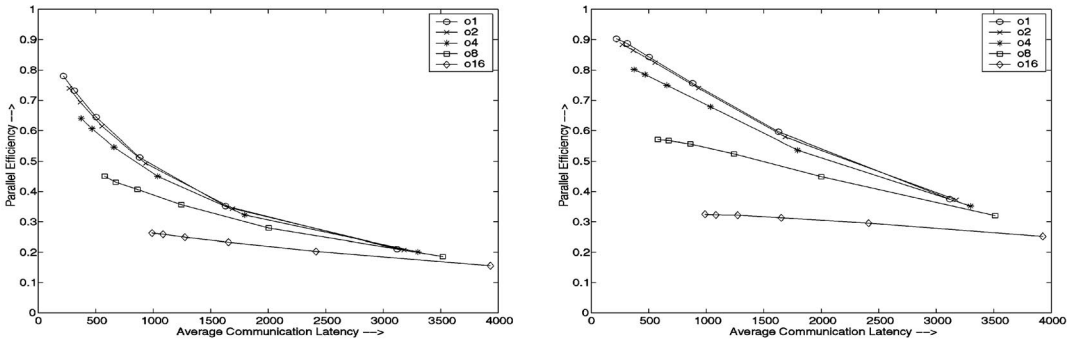


Fig. 9. (a) Nonprefetched and (b) prefetched Radix-Sort running on 64 processors and 2M keys.

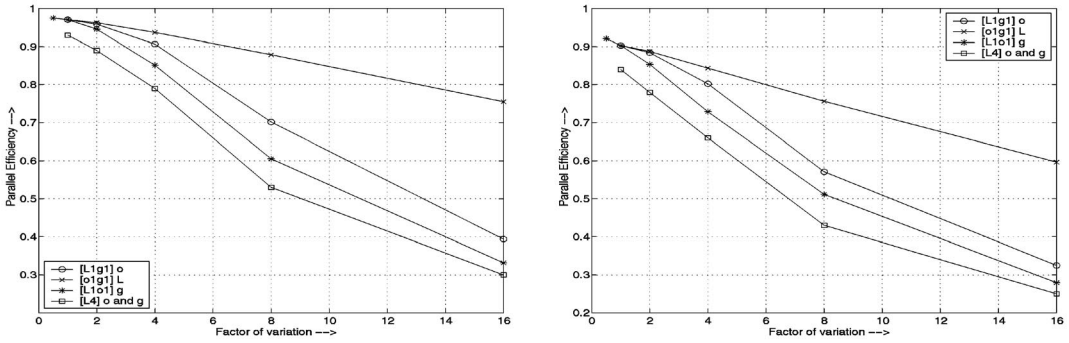


Fig. 10. Effect of varying l, o, and g on prefetched 2M-key Radix-Sort for (a) 32 and (b) 64 processors.

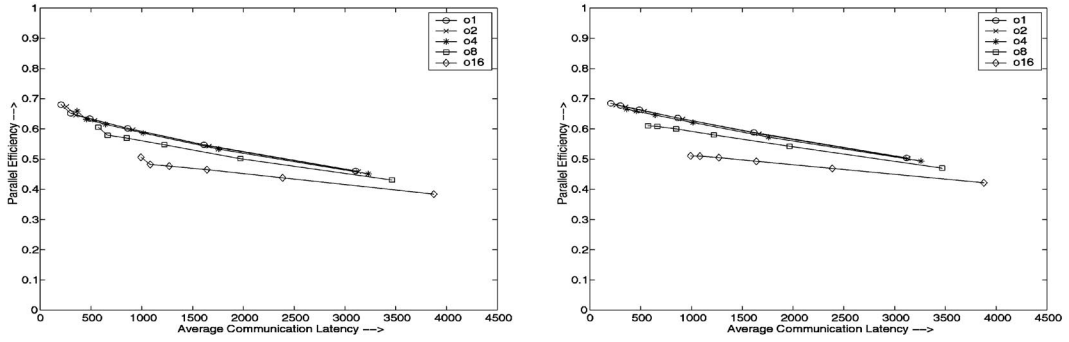


Fig. 11. (a) Nonprefetched and (b) prefetched LU running on 64 processors and a 512×512 matrix.

Radix-Sort, which requires all-to-all communication consisting of bursty writes. Also, the average bandwidth requirement for Radix-Sort is the maximum among our six applications [13]. The combined effect of o and g shows that the combined controller-link bandwidth is still the most important determinant of performance (for 64 processors, $L_{16}O_1g_1$ achieves a parallel efficiency of 0.6, while $L_4O_{16}g_{16}$ achieves an efficiency of only 0.25). Again, the same trend continues to hold for network latency: It is less important than o and g.

LU: The efficiency curves for LU are presented in Fig. 11. One significant difference for both prefetched and nonprefetched LU is that the performance is less sensitive to both latency and occupancy. The reason is that LU has a low communication-to-computation ratio, and the dominant bottleneck in such high-performance matrix factorizations is load imbalance, so its performance is less dependent on communication costs. The effect of varying g was similar to Radix-Sort: Network latency remains insignificant compared to combined controller-link bandwidth.

Barnes-Hut and Water: Fig. 12 shows the results of Barnes-Hut for 8,192 bodies (although the simulation was run over three time steps, the speedup numbers are measured for the last time step only) and Water for 1,024 molecules. Neither application includes prefetching because the high degree of temporal locality (and irregularity in Barnes-Hut) makes it difficult to determine which particular memory references will miss in the cache. For

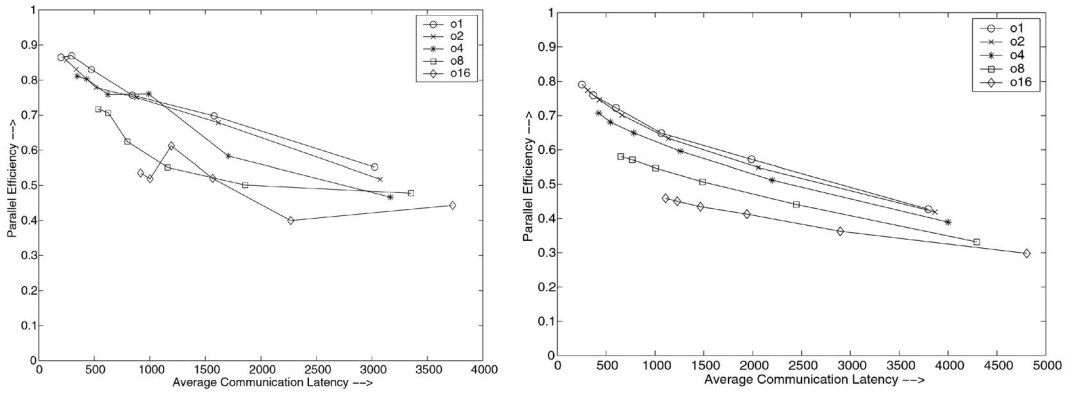


Fig. 12. (a) Barnes-Hut (8,192 bodies) and (b) Water (1,024 molecules) running on 64 processors.

Barnes-Hut, the O_1 and O_2 controllers have almost identical performance. For some values of o (e.g., 16), increasing the network latency sometimes increases the performance. We observed that this anomaly happens because of reduced synchronization stall time with a slow network. With a relatively slow network and for some particular event timing the lock accesses from different processors may become nicely staggered in time leading to lower lock contention. For Water, the efficiency curves are similar to those for LU. But, Water shows higher sensitivity to latency and occupancy than LU. For Barnes-Hut with a reduced problem size (4,096 bodies), [not shown] there was not as significant performance loss as in Water with smaller problem size (512 molecules) [not shown] for fast controllers (O_1, O_2, O_4). For Water with 512 molecules, there was a significant performance loss. The parallel efficiency achieved by an O_1 controller for all latencies with 512 molecules was consistently less than that achieved by an O_8 controller running on 1,024 molecules. The graphs for these results are not presented here because of space constraints, but are available in our technical report [4]. The gap parameter g was not found to be important for either application.

Application Summary: As we expected, increasing network latency uniformly decreases overall performance across all the applications. Prefetching is often very effective at improving performance, but requires low occupancy controllers. Also, we observed that controller occupancy is much more important than network latency. For some of the applications (e.g., Radix-Sort), the node-to-network bandwidth is more important than the controller occupancy for fast controllers (e.g., hardwired). But, we do not expect the node-to-network bandwidth to ever become a bottleneck because fast controllers are expected to have fast network interfaces. In other words, the node-to-network bandwidth may become a bottleneck in certain applications with bursty communication phases if the message bandwidth of the controller is not well-balanced with the bandwidth of the network interface. We noticed that the contention effect of controller occupancy 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 O_4 or O_8, and by O_{16} (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. The only hope for less aggressive controllers is that larger problem sizes will restore some lost parallel efficiency. We explore this possibility via experimentation on a DSM prototype in Section 6.

5 Analytical Modeling

In this section, we develop a mathematical model to further understand the impact of latency and occupancy-induced contention on the execution time of an application. We show that it is easy to model the average communication latency, but extremely difficult to predict how contention varies across our design space.

Let the execution time for the L_1O_1 model be t_1 and that for the L_xO_y model be t_2. We expect that

$$t_2 = t_1 + \overline{V}_T(\delta T_L + \delta T_C), \tag{2}$$

where $\overline{V}_T$ is the per-node average transaction volume, δT_L is the average change in uncontended transaction latency, and δT_C is the average change in communication controller contention per protocol transaction. If we want to predict t_2 from t_1, we need three parameters, namely, $\overline{V}_T$, δT_L, and δT_C. We explore each of these parameters separately.

5.1 Modeling $\delta \mathbf{T_L}$

We can predict δT_L for prefetched FFT within 2 percent of our simulation results in most cases. We can achieve similar accuracy for the other five applications as well. From the detailed L_1O_1 simulation of prefetched FFT, we find that the average transaction latency is given by the equation

$$T_L = 1.42l + 3o + 51 \tag{3}$$

in processor clock cycles. Since the transaction volume and transaction pattern remain more or less unchanged as l and o are varied, we expect that this equation holds even for values of x and y other than 1. Table 5 shows the validity of this equation as we try to predict the average transaction latency (in processor clock cycles) using this equation, and compare them against the real values obtained through simulation. As can be seen from the table, the predicted values are, in most cases, within 1 percent of the values

TABLE 5
Predicting Average Communication Latency

Config.	Simulated	Predicted	Config.	Simulated	Predicted
L1O1	164	164	L8O4	795	787
L2O1	240	235	L16O4	1371	1355
L4O1	384	377	L32O4	2522	2491
L8O1	672	661	L1O8	463	458
L16O1	1246	1229	L2O8	534	529
L32O1	2394	2365	L4O8	677	671
L1O2	211	206	L8O8	963	955
L2O2	282	277	L16O8	1535	1523
L4O2	426	419	L32O8	2685	2659
L8O2	713	703	L1O16	798	794
L16O2	1289	1271	L2O16	870	865
L32O2	2437	2407	L4O16	1012	1007
L1O4	294	290	L8O16	1310	1291
L2O4	366	361	L16O16	1871	1859
L4O4	509	503	L32O16	3019	2995

obtained from simulation. Since δT_L is simply the difference between two average latencies, we can predict that within 2 percent of error. We must note that (3) is application dependent, but, for each application, we need to run only one simulation to predict δT_L for the whole design space (30 points with fixed g, P, and problem size). Having predicted δT_L with high accuracy, we concentrate on the remaining two parameters.

5.2 Modeling $\overline{\mathbf{V}}_{\mathbf{T}}$

The major problem in measuring the exact volume of protocol transactions is that some of these transactions are hidden under computation, while some of these mutually overlap in time. The former problem is more pronounced in prefetched applications, while the latter one introduces double-counting. We want to measure the volume of transactions that do not overlap with computation (i.e., are not hidden), and we also want to avoid double-counting. We developed two methods for doing this.

Our original study took into account only remote read misses to measure the volume of transactions. The first model we examine is an obvious extension to that. We take into account the dominant transaction type to measure $\overline{V}_T$. For example, for FFT, it is remote read misses and, for Radix-Sort, it is local read misses that are dirty on a remote node. But, this method, when combined with our prediction of δT_L and δT_C, exhibits poor accuracy in predicting the overall parallel efficiency, and we do not pursue it further.

The second method is more involved and requires two simulations to calculate $\overline{V}_T$. Since we want to reduce the amount of mutual overlap between transactions and the amount of overlap between computation and communication, we calculate $\overline{V}_T$ from the $L_{16}O_1$ and $L_{32}O_1$ simulations. With a slow network, we expect that the amount of overlap between transactions and computation will be reduced. From these two simulations, we calculate δT_L and δT_C between these two configurations and use (2) to calculate $\overline{V}_T$. This method leads to much better overall prediction accuracy. The results are presented in the following section.

5.3 Modeling $\delta \mathbf{T}_{\mathbf{C}}$ and Overall Prediction

Accurately modeling contention in any kind of centralized or distributed controller is a big challenge. This is simply because contention can arise due to various direct or indirect effects in the underlying architecture. The direct effects come from the service rate of the controller and the arrival rate of the requests. The indirect effects depend on how different types of requests and replies interact with each other, usually in a very unpredictable manner, as the system operates over time. The obvious way to reason and understand about the contention in a controller is to see it as a centralized queue leading to a server.

We model the node controller as an $(\mathrm{M/M/1}) : (\mathrm{FCFS}/\infty/\infty)$ queuing system. Although the queues in the real system are of finite capacity, we model them as infinite queues. We shall comment on this assumption later. We assume Markovian arrival and departure as it turns out that this model predicts overall performance well. Since each node has a single communication controller, the queuing model has a single server with a First-Come-First-Served service policy. Also, note that the backpressure flow-control is not taken into account while modeling the controller and only the effects of l and o are analyzed for fixed values of g and P. The backpressure flow-control, which is present in any realistic closed system, mainly arises due to the fact that the main processor can issue only a finite number of requests. So, very high contention in the controller will eventually lead to a decrease in the arrival rate, especially for nonprefetched applications. To model backpressure flow-control, one needs to consider a finite population of requests instead of the infinite calling source analysis presented here. But, the main problem in modeling a finite population is the determination of the exact size of the population. It may appear that the size of the population should be simply the total number of processors multiplied by the maximum number of outstanding misses in each processor. But, this value strictly under-estimates the population because a communication controller may generate special coherence messages (e.g., invalidation requests, intervention messages, i.e., forwarded requests, ownership transfer messages, sharing writebacks, etc.). So, we carry out an infinite calling source analysis and

show that this simple model is accurate enough in predicting the contention in the communication controller. Another reason for using a simple model is that we wanted (if possible) to devise an easy way to predict the parallel efficiency.

The contention in the system is measured as the expected waiting time in the queue, W_q, which is given by [29]:

$$W_q = \frac{\rho}{\mu(1-\rho)}, \tag{4}$$

where ρ is the effective traffic in the system and is given by $\rho = \lambda/\mu$, λ being the arrival rate (number of requests arriving per processor clock cycle) and μ the service rate (number of requests serviced per processor clock cycle). We assume that the arrival rate λ is inversely proportion to the network latency, i.e., for an L_xO_y model $\lambda = K_L/x$, where K_L is a system and application dependent constant. Similarly, we assume that the service rate $\mu = K_O/y$. Thus, we have

$$W_q = \frac{K_L y^2}{K_O(K_O x - K_L y)}. \tag{5}$$

Observe that, for fixed occupancy (i.e., fixed y), as latency increases (i.e., x increases), contention decreases, which is the expected result. From the contention in the L_xO_1 model we try to predict that in L_xO_y model. Let W_{q1} and W_{q2} be the contention in the L_xO_1 model and L_xO_y model, respectively. Therefore, we have

$$W_{q2} = y^2\left(\frac{K_O x - K_L}{K_O x - K_L y}\right)W_{q1}. \tag{6}$$

This equation directly follows from (5). It is interesting to note that growth in contention (i.e., W_{q2}/W_{q1}) for fixed latency (i.e., x) is faster than quadratic in occupancy because, for $y > 1$, the bracketed term in (6) is bigger than 1 and this term increases in magnitude as x (i.e., latency) is kept fixed and y (i.e., occupancy) is increased. Further, note that, for fixed y (i.e., fixed occupancy), the ratio W_{q2}/W_{q1} does not vary much with x (i.e., latency). So, our claim that occupancy affects contention more than latency does is born out analytically. We use (6) to predict the contention of L_xO_y configuration from that in L_xO_1 configuration as y is varied and x is kept fixed. To carry out this prediction, we only need to carry out the six simulations (versus the 30 simulations that cover our entire design space) for the L_xO_1 model as x takes on values 1, 2, 4, 8, 16, and 32. From these simulation results, we can calculate the contention for these six models. The next task is to estimate K_L and K_O. We approximate the arrival rate as the total volume of transactions per communication controller divided by the parallel execution time. From this, we calculate K_L. Next, we approximate the service rate as the reciprocal of communication controller occupancy per protocol handler invocation. From this, we calculate K_O.

Using the Analytical Model: For architectures with fast networks (i.e., low values of x), if the controller is relatively slow (i.e., relatively bigger values of y), this model will mispredict contention. Whether it will underpredict or overpredict contention depends on the communication structure of the underlying application. The main reason for this misprediction is that the fast network exposes the finite queue length limitation and the system behaves more like an $(\mathrm{M/M/1}) : (\mathrm{FCFS/N/\infty})$ model, where N is the maximum

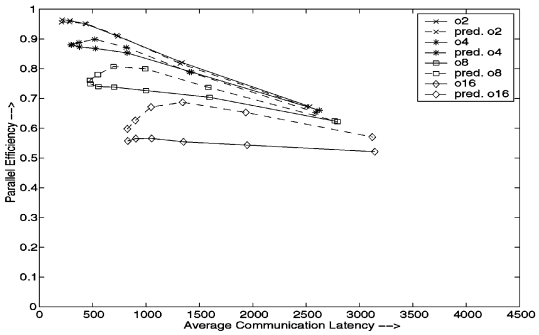


Fig. 13. Prediction accuracy of the linear model for prefetched FFT on 64 processors.

number of outstanding requests in the system. At this point, a linear waiting time model (for fixed x, e.g., for L_1 network, the contention increases linearly with y) works quite well. To be more precise, in the linear model the contention for L_xO_2 configuration is exactly double that of L_xO_1 configuration for fixed x. For prefetched FFT, the linear waiting time model predicts the parallel efficiency within 1 percent error for L_1O_2, L_2O_2, L_1O_4, L_2O_4, L_1O_8, and within 5 percent for L_1O_{16} configurations (see Fig. 13; the simulated curves are in solid lines, while the predicted ones are in dotted lines). $\overline{V}_T$ is predicted by the second method described in Section 5.2. However, we observed that the prediction error of the linear model for prefetched Radix-Sort is as big as 5 percent to 10 percent for the low-latency configurations (interested readers are refered to [4]). The reason for this is mainly the lack of knowledge about the exact $\overline{V}_T$. With low-latency networks, prefetching can hide latency, and our model mispredicts $\overline{V}_T$, especially when the communication structure is irregular and bursty. Our experience says that different queuing models are necessary to predict the performance of different applications due to the vastly different communication structures of different applications. Our dual queuing model (to be introduced shortly) predicts the parallel efficiency of our applications within a small percentage for all our configurations.

We now present a mathematical proof to explain why a linear waiting time model works well for the low-latency network configurations. As we have already pointed out. for these configurations the system behaves more like an $(\mathrm{M/M/1}) : (\mathrm{FCFS/N/\infty})$ model. For this model, the expected system queue length is given by [29]:

$$L_s = \frac{\rho[1-(N+1)\rho^N + N\rho^{N+1}]}{(1-\rho)(1-\rho^{N+1})}. \tag{7}$$

The average queue length is given by

$$L_q = L_s - \frac{\lambda_{\mathrm{eff}}}{\mu}, \tag{8}$$

where effective arrival rate, λ_{eff} is given by

$$\lambda_{\mathrm{eff}} = \lambda\left[1 - \left(\frac{1-\rho}{1-\rho^{N+1}}\right)\rho^N\right]. \tag{9}$$

For a fast network and a slow controller, it is reasonable to assume that ρ is large. So, for reasonable values of N (e.g., 16 for the PI inbound queues in our simulator), it is logical

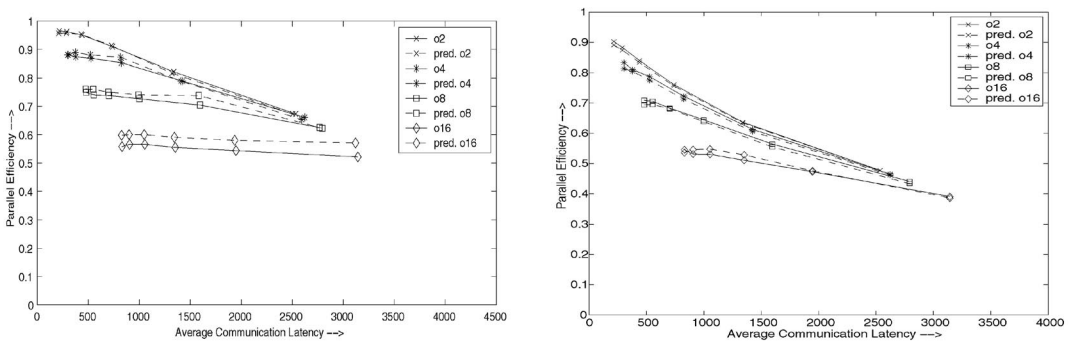


Fig. 14. (a) Prediction accuracy of prefetched FFT with the DSM dual queuing model and (b) prediction accuracy of nonprefetched FFT with the linear model.

to assume that $1/\rho^{N+1}$ is negligible. With these simplifications, we obtain

$$\lambda_{\text{eff}} = \lambda\left[1 - \left(\frac{1/\rho - 1}{-1}\right)\right], \tag{10}$$

i.e., $\lambda_{\text{eff}} = \mu$, which is expected because now the arrival rate is governed by the service rate since the queue remains full most of the time and there is no space to accommodate new requests until a pending request gets serviced. Therefore,

$$W_q = \frac{L_q}{\lambda_{\text{eff}}}, \tag{11}$$

which, from (8) and the relation $\lambda_{\text{eff}} = \mu$, reduces to

$$W_q = \frac{1}{\mu}(L_s - 1) = \frac{1}{K_O}(L_s - 1)y. \tag{12}$$

Now, for a fixed network model (i.e., fixed x), it is reasonable to assume that L_s remains constant as y is varied. This assumption is justified by the fact that for a fast network and relatively slow controller, L_s is almost always close to N and it is more affected by the arrival rate than by the service rate since the service rate is much smaller than the arrival rate. We present an approximate analysis to support this view. In (7), assuming that ρ is large, we obtain

$$L_s = \left(\frac{\rho}{\rho - 1}\right)\left(N - \frac{N+1}{\rho}\right), \tag{13}$$

which we can approximate to

$$L_s = N - \frac{N+1}{\rho} = N - \frac{N+1}{\lambda}\mu, \tag{14}$$

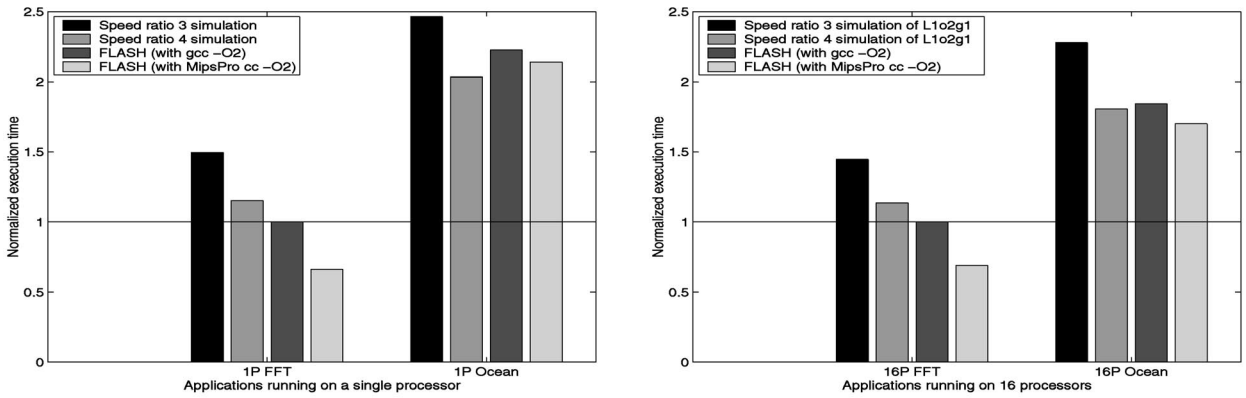
with the assumption that $\rho >> 1$. Now, if λ is large, then $(N+1)/\lambda$ is small, and multiplying it by a small μ will not change L_s much as we decrease μ from L_1O_1 to L_1O_{16}. Thus, (12) precisely describes the linear waiting time model—the contention varies linearly with occupancy factor y for fixed x. This completes our proof. Also, note that, for a fixed occupancy, as latency increases, the arrival rate decreases and L_s also decreases leading to less contention. But, this equation fails to hold as latency increases beyond that of an L_2 network because this model dictates a faster decrease in contention than actually occurs in practice.

As the network gets slower, finite queue length is no longer a problem and our original quadratic contention model (6) works quite well. For example, in prefetched FFT, we found that this model predicts the parallel efficiency within 6 percent error for L_4O_4, L_4O_8, L_8O_8, L_2O_{16}, L_4O_{16}, L_8O_{16}, and $L_{16}O_{16}$ configurations (see Fig. 14a). To predict parallel efficiency for the other points, Fig. 14a uses the linear waiting time model. The combination of these two models give rise to a hybrid queuing model, which we call the *DSM dual queuing model*.

As the network gets even slower, a completely new phenomenon takes over. Now, network contention comes into play and the outbound queue length becomes a bottleneck. As the outbound queues fill up, the controller stalls more frequently and is unable to send out messages. As a result, the service rate gets affected and the input queues start filling up, once again exposing the finite length of inbound queues. Again, the linear contention model works well, as can be seen from Figs. 13 and 14.

The failure of the linear waiting time model to predict the efficiency for Radix-Sort at low latency indicates a major problem in modeling the contention of a CC-NUMA system. The difficulty is in perfectly calculating the volume of protocol transactions that really cannot be hidden under computation. Also, a problem arises in using only the transactions that do not overlap in time so that we avoid double-counting. However, for nonprefetched FFT, we observed that the linear waiting time model predicts the efficiency curves quite well (see Fig. 14b). The maximum prediction error is 1 percent. This is expected since contention in the communication controller will be less without prefetching than with prefetching. But, still, for O_{16}, the linear waiting time model under-predicts the contention for moderate values of l. This is why we observe higher values of predicted efficiency for L_4O_{16} and L_8O_{16} as compared to the simulated efficiency. This means that even without latency hiding techniques, the growth rate of contention tends to be faster than linear in occupancy as the communication controller moves more toward commodity microprocessors on the memory or I/O bus.

Finally, we summarize our findings about modeling the contention in the communication controller. The system switches between two models as we traverse the design space and we call it the DSM dual queuing model. The exact points where the system moves from one model to another are highly dependent on the communication structure of the running application. But, our experience says that, for O_4, O_8 and O_{16} controllers, the system switches from the



(a) (b)

Fig. 15. (a) Calibration of uniprocessor simulations against FLASH and (b) locating FLASH in the design space.

linear model to the quadratic model around L_2 network latency and reverts back to the linear model at L_{32} network latency and above. Overall, contention is much more important than latency since the latter scales only linearly with l and o.

6 Varying Occupancy on a DSM Prototype

In this section, we present the effects of varying occupancy for larger problem sizes (that we could not simulate). Normally, it is not possible to increase occupancy in a DSM machine once it is built. However, we have the luxury of using the programmable protocol processor of a 16 and 32-node Stanford FLASH multiprocessor for this purpose. The 75 MHz communication controller has an embedded RISC protocol processor that runs software handlers to satisfy protocol requests. The main processor is a 225 MHz MIPS R10000. Although we can vary occupancy, we cannot vary l and g on FLASH because the message send/receive mechanism in FLASH is hard-wired and integrated into the network interface.

6.1 Locating FLASH in the Design Space

To properly interpret the results obtained from the programmable protocol engine, we first try to locate FLASH in our design space using the parallel efficiency obtained from a 16-node FLASH multiprocessor. FLASH uses an SGI Spider router [8] that takes eight network cycles to get the header through the chip. The data payload, if any, follows pipelined at four bytes per network clock. The network clock speed of the current FLASH prototype is 150 MHz. Thus, the peak node-to-network bandwidth is 600 MB/s. Since the communication controller (MAGIC) is running at 75 MHz, the network hop time is four system cycles, i.e., approximately 50ns. In our design space, for a 64-processor mesh topology, the L_1 network latency is 25 system cycles. Since the average number of hops for a mesh topology on 64 nodes is 5.33, the average per-hop time is approximately five system cycles, i.e., 50ns at 100 MHz system clock frequency. Therefore, the FLASH network is close to L_1 and the node-to-network interface is also greater than g_1 (which is 400 MB/s), though we have seen that this does not improve performance for our applications. To figure out the occupancy of FLASH, we ran simulations for prefetched FFT and Ocean with our base problem sizes on a 16-node configuration with processor/memory speed ratios of 3 and 4, and tried to map the corresponding results from FLASH onto our simulation results. While doing this comparison, we had to be careful in our choice of compiler. For a target architecture of an R10000, the MipsPro cc compiler produces better code than gcc. But, since our simulation environment uses gcc to compile the applications, we used the same compiler for compiling the applications for FLASH that we use in this comparison. Fig. 15a shows the normalized execution times of FFT (1M points) and Ocean (514×514 grid) on a single processor. All times are normalized to uniprocessor FLASH execution time of FFT with gcc compiler. A comparison between the simulations with speed ratio 4 and FLASH gcc results tells us that a speed ratio of 4 in our simulations models the FLASH machines fairly well. This was also found to be true in [10]. Using a speed ratio of 4 on 16 processors to locate FLASH in our design space, we find that $L_1O_2g_1$ simulation results are closest to the 16 processor FLASH results. To see how well they match, we present the results for FFT (1M points) and Ocean (514×514 grid) in Fig. 15b. All times are again normalized to 16 processor FLASH execution time of FFT with gcc compiler. Thus, we have calibrated FLASH as an $L_1O_2g_1$ system in our simulation study using the same compilers. But, in the results presented below, to show how occupancy affects performance on FLASH, we use the MipsPro cc compiler with O2 optimization because it produces better code than gcc.

6.2 Effect of Increasing Occupancy

Since FLASH is a working DSM prototype, we can run bigger problem sizes than we can simulate and, therefore, we can examine the performance of controllers with large occupancy at these larger problem sizes. We are able to model higher occupancy machines on FLASH by increasing the occupancy of the handlers run by the programmable protocol processor. To vary the occupancy of the communication controller, we doubled the occupancy of each protocol handler at every step by inserting the appropriate number of NOPs in the handler code. We checked that the communication controller instruction cache miss rate remains unchanged for the instrumented code and does not cloud our results. Figs. 16 and 17 show the results for

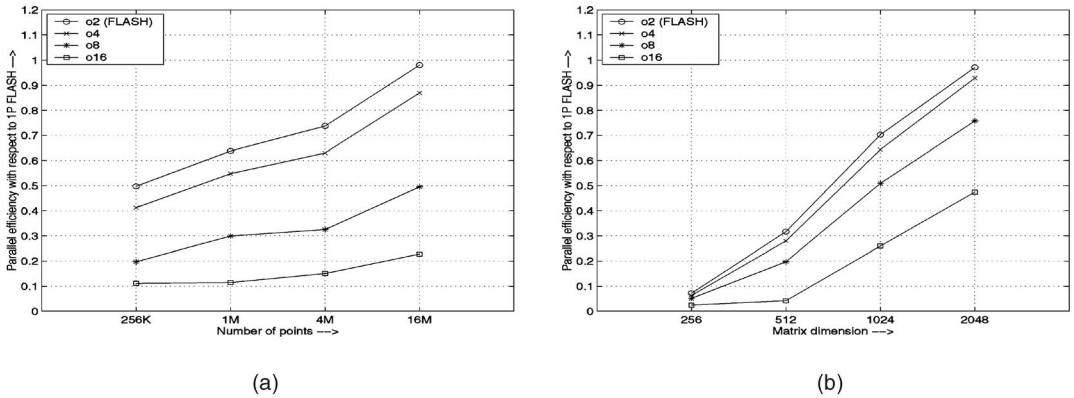


Fig. 16. (a) Prefetched FFT and (b) prefetched LU on 32-processor FLASH.

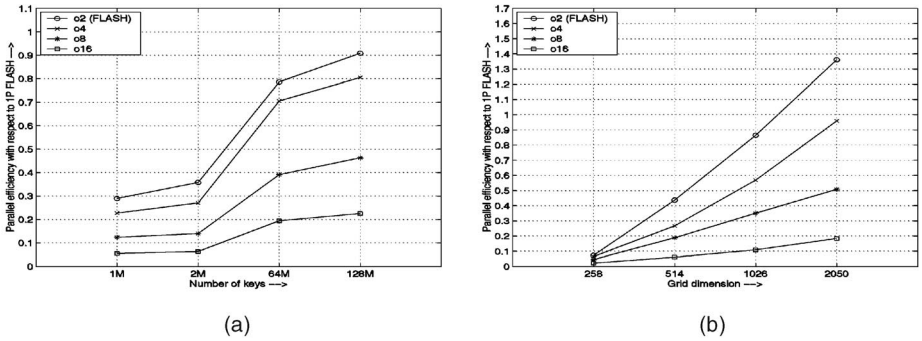


Fig. 17. (a) Prefetched Radix-Sort and (b) prefetched Ocean on 32-processor FLASH.

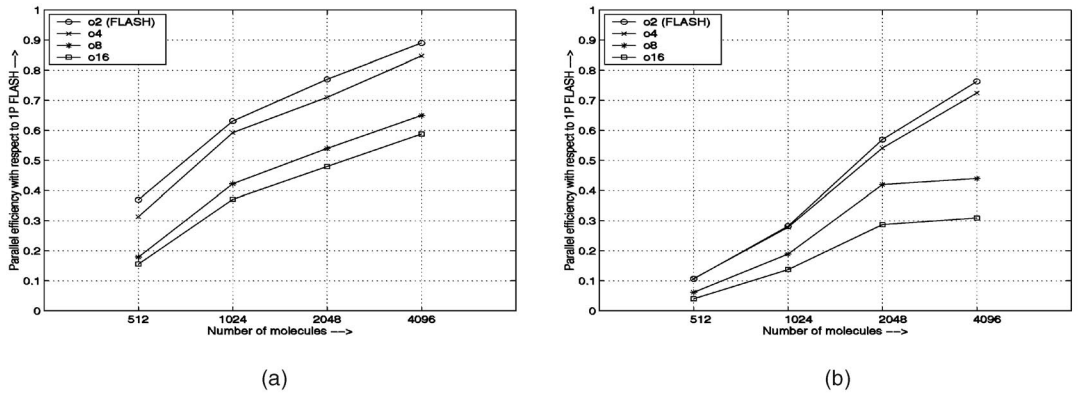


Fig. 18. Water on (a) 16 and (b) 32-processor FLASH.

FFT, LU, Radix-Sort, and Ocean on a 32-node FLASH. Fig. 18 shows the results for Water on both a 16 and 32-node FLASH. Due to shortage of space, we do not show the graphs of 16-node runs for FFT, LU, Radix-Sort, and Ocean. We only mention the salient observations from these runs. Interested readers are refered to [4].

Although prefetched FFT scales well up to an O_8 controller on 16 processors (not shown), only O_2 and O_4 controllers scale well for 32 processors (Fig. 16a). On 16 processors, an O_8 controller achieves a parallel efficiency of 0.66 with 16M points, while on 32 processors, the parallel efficiency is only 0.5. Also, the O_{16} controller with 16M points fails to achieve even the efficiency achieved by FLASH on 256K points for both the processor counts. A careful examination of the slopes of the O_8 and O_{16} curves clearly tells us that the performance gap between these two controllers will continue to increase (the curves diverge) as we keep quadrupling the problem size. This, in turn, means that, to regain the lost performance on an O_{16} controller, we need an extremely fast growth rate in problem size. FFT has a computation time of $O(n \log n)$ and a communication volume of $O(n)$, where n is the number of points. Thus, the computation-to-communication ratio is $O(\log n)$, which clearly increases with problem size. But, just as in many structured applications, communication in FFT 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.

Prefetched LU (Fig. 16b) shows some interesting properties which are not present in FFT. In FFT, we see a big drop in

performance as we go from an O_4 to an O_8 controller, indicating that there is a minimum level of controller performance necessary to achieve good performance. But, for LU this is not the case. A comparison of the slopes of the curves in FFT and LU tells us that occupancy is not as big a problem for LU as for FFT, which, in turn, supports our simulation findings. LU requires a much smaller growth rate in problem size as compared to FFT. But, still, in a 32-node system, LU's performance increases relatively slowly with increasing problem sizes for an O_{16} controller as the controller starts to saturate. The slower growth rate in problem size for LU as compared to FFT is explained by the fact that LU has a linear growth rate in computation-to-communication ratio (computation of $O(n^3)$ and communication of $O(n^2)$), while FFT has only a logarithmic growth rate in computation-to-communication ratio.

The performance of prefetched Radix-Sort (Fig. 17a) has a striking similarity to that of FFT. However, a careful examination of the problem sizes of Radix-Sort actually tells us that its performance is much worse than FFT. Its performance is not very encouraging for high-occupancy controllers. Even with a problem size of 128M keys, the O_{16} controller achieves an efficiency of only 0.3 on 16 processors and 0.23 on 32 processors. It is difficult to scale Radix-Sort because of its bursty write behavior [31]. As the problem size grows, these writes also tend to be remote, which, in effect, doubles the number of protocol transactions and leads to excessive contention. Also, the irregular communication pattern causes hot-spots in the memory system, resulting in even worse performance. Finally, Radix-Sort has a constant overall computation-to-communication ratio.

Prefetched Ocean (Fig. 17b) scales well up to an O_4 controller, but the efficiency curve has a very small slope for an O_{16} controller. At O_{16}, Ocean achieves a parallel efficiency of only 0.3 for a grid size of $2,050 \times 2,050$ on 16 processors and 0.19 on 32 processors. We continue to see a big gap between O_4 and O_8 curves, in addition to the big performance gap between O_8 and O_{16} curves. Although Ocean has a linear growth rate in computation-to-communication ratio, it also communicates data in structured phases leading to a constant computation-to-communication ratio within those phases. One final scaling problem in Ocean is that 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.

Water scales well for all controller architectures on 16 nodes (Fig. 18a), but we observe a clear saturating trend in performance for O_8 and O_{16} controllers on 32 nodes (Fig. 18b). This result helps us establish the fact that, as the system moves toward even larger DSM multiprocessors, controller occupancy will become even more important for parallel performance.

Overall, significant increases in problem size are necessary for the less aggressive controllers to achieve the desired efficiency on 16 and 32-processor systems with a fast MPP network. There are many important classes of applications (transform methods, sorting, multigrid equation solver) 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 controller played an important role in determining the required growth in problem size. We observed a general trend that all applications scaled reasonably well up to an O_4 controller. But, for an O_{16} controller, none of the applications showed promising performance on a 32-node system. As we scale the number of processors further, we expect similar subpar performance for O_8 controllers. Therefore, as the network becomes slower and the system grows toward even larger DSM multiprocessors, we expect that only the more tightly integrated, aggressive communication controllers will achieve acceptable DSM performance, regardless of problem size.

7 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. Through simulation, analytical modeling, and experimentation on a flexible DSM prototype, we evaluated the performance impact of latency, occupancy, node-to-network bandwidth, and processor count over a range of representative scientific applications. Our results showed 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. As expected, network latency impacted overall performance, but its importance diminished with high-occupancy controllers, or when applications employed latency hiding mechanisms. We also observed that, for a fast hardwired controller or a customized coprocessor used as the communication controller, node-to-network bandwidth can become important, especially for applications with bursty communication phases. Stated differently, for these applications the controller message bandwidth should be well-balanced with the link-bandwidth of the interface so that neither becomes a bottleneck. However, the node-to-network interface speed is typically related to the controller speed, therefore, we do not expect the node-to-network bandwidth to be a problem for aggressive controller designs.

Our main result is that the occupancy of the communication controller is critical to good performance in DSM machines and, in most cases, is more important than both network latency and node-to-network bandwidth. For machines with tightly coupled MPP networks, we found that controller occupancy has a large performance impact regardless of whether or not applications incorporated latency hiding techniques. For machines with loosely-coupled networks, we showed that while without latency hiding occupancy did not matter to overall performance, 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. Recalling that controller occupancy is the reciprocal of controller bandwidth (in messages), we found that it was easier to hide network latencies than it was to overcome communication bandwidth shortage.

Moreover, it was not the latency component of the higher occupancy controllers that caused performance degradation, but rather the contention component, even without

latency hiding. We introduced a DSM dual queuing model to analytically describe this contention. This model showed that the growth rate of contention is more than quadratic in occupancy for moderate-latency networks (e.g., distributed MPP and fast LANs). Thus, our model showed analytically that occupancy is more important than latency because of its impact on contention in the system.

Finally, a thorough study on a real DSM machine showed that for many classes of applications it is extremely difficult for architectures with higher values of controller occupancy to achieve high parallel efficiency. The problem sizes needed to achieve high parallel efficiency quickly become unreasonable. For the applications we have considered here, it is impossible to regain the lost performance as one moves beyond hardwired and customized controllers and more toward general purpose microprocessors on the memory or I/O bus. On the other hand, the occupancies of specialized or hardwired controllers on the memory bus were low enough to achieve good efficiency for all the applications in this study.

The tendency among DSM designers has been to focus on latency and network bandwidth as the important performance issues in the communication architecture. Our results demonstrate that the occupancy of the communication controller is the most important architectural parameter that affects the parallel performance of a DSM multiprocessor and that the message bandwidth of the controller should be well-balanced with the link bandwidth of the network interface to achieve good parallel performance.

REFERENCES

[1] A. Agarwal et al. "The MIT Alewife Machine: Architecture and Performance," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 2-13, June 1995.

[2] A. Bilas and J.P. Singh, "The Effects of Communication Parameters on End Performance of Shared Virtual Memory Clusters," *Proc. 1997 Supercomputing Conf. High Performance Networking and Computing,* Nov. 1997.

[3] M.A. Blumrich et al. "A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 142-153, Apr. 1994.

[4] M. Chaudhuri et al. "Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation," Technical Report CSL-TR-2002-1025, Computer Systems Laboratory, Cornell Univ., Ithaca, NY 14853. Available at http://www.csl.cornell.edu/TR/CSL-TR-2002-1025.ps, July 2002.

[5] F.T. Chong et al. "The Sensitivity of Communication Mechanisms to Bandwidth and Latency," *Proc. Fourth Int'l Symp. High Performance Computer Architecture,* pp. 37-46, Feb. 1998.

[6] D. Culler et al. "LogP: Toward a Realistic Model of Parallel Computation," *Proc. Fourth Symp. Principles and Practice of Parallel Processing,* pp. 1-12, May 1993.

[7] D. Dai and D.K. Panda, "Building Efficient Limited Directory-Based DSMs: A Multidestination Message Passing Based Approach," Technical Report OSU-CISRC-4/96-TR21, Dept. of Computer and Information Science, Ohio State Univ., Columbus, OH 43210-1277, 1996.

[8] M. Galles, "Spider: A High-Speed Network Interconnect," *IEEE Micro,* vol. 17, no. 1, pp. 34-39, Jan.-Feb. 1997.

[9] K. Gharachorloo et al. "Architecture and Design of AlphaServer GS320," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 13-24, Nov. 2000.

[10] J. Gibson et al. "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 49-58, Nov. 2000.

[11] M. Heinrich et al. "The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 274-285, Oct. 1994.

[12] M. Heinrich and E. Speight, "Providing Hardware DSM Performance at Software DSM Cost," Technical Report CSL-TR-2000-1008, Computer Systems Lab., Cornell Univ., Ithaca, NY 14853, 2000.

[13] C. Holt et al. "The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors," Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford Univ., Stanford, CA 94305, 1995.

[14] J. Kuskin et al. "The Stanford FLASH Multiprocessor," *Proc. 21st Int'l Symp. Computer Architecture,* pp. 302-313, Apr. 1994.

[15] Kendall Square Research, "KSR1 Technical Summary,"technical report, Waltham, MA, 1992.

[16] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 241-251, June 1997.

[17] D. Lenoski et al. "The Stanford DASH Multiprocessor," *IEEE Computer,* vol. 25, no. 3, pp. 63-79, Mar. 1992.

[18] R.P. Martin et al. "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 85-97, June 1997.

[19] M. Michael at al. "Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 219-228, June 1997.

[20] A.K. Nanda et al. "High-Throughput Coherence Controllers," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture,* pp. 145-155, Jan. 2000.

[21] C.C. Niessen and D.G. Meyer, "High Performance Network Interfaces," *Proc. First Midwest Workshop Parallel Processing,* Aug. 1999.

[22] A. Nowatzyk et al. "The S3.mp Scalable Shared Memory Multiprocessor," *Proc. 24th Int'l Conf. Parallel Processing,* pp. 1-10, Aug. 1995.

[23] S.K. Reinhardt, R.W. Pfile, and D.A. Wood, "Decoupled Hardware Support for Distributed Shared Memory," *Proc. 23rd Int'l Symp. Computer Architecture,* pp. 34-43, May 1996.

[24] E. Rothberg, J.P. Singh, and A. Gupta, "Working Sets, Cache Sizes, and Node Granularity for Large-Scale Multiprocessors," *Proc. 20th Int'l Symp. Computer Architecture,* pp. 14-25, May 1993.

[25] I. Schoinas et al. "Fine-Grain Access Control for Distributed Shared Memory," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 297-306, Oct. 1994.

[26] J.P. Singh et al. "Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole and Radiosity," *J. Parallel and Distributed Computing,* vol. 27, no. 2, pp. 118-141, June 1995.

[27] D.J. Sorin et al. "Analytic Evaluation of Shared-Memory Systems with ILP Processors," *Proc. 25th Int'l Symp. Computer Architecture,* pp. 380-391, June 1998.

[28] C.B. Stunkel et al. "The SP2 High-Performance Switch," *IBM Systems J.,* vol. 34, no. 2, pp. 185-204, Feb. 1995.

[29] H.A. Taha, *Operations Research: An Introduction,* sixth ed. 1996.

[30] D.A. Wood and M.D. Hill, "Cost-Effective Parallel Computing," *IEEE Computer,* vol. 28, no. 2, pp. 69-72, Feb. 1995.

[31] S.C. Woo et al. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 24-36, June 1995.

[32] S.C. Woo, J.P. Singh, and J.L. Hennessy, "The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 219-229, Oct. 1994.

[33] Y. Zhou et al. "Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation," *Proc. Sixth Symp. Principles and Practice of Parallel Programming,* pp. 193-205, June 1997.

[34] Z. Zhou, W. Shi, and Z. Tang, "A Novel Multicast Scheme to Reduce Cache Invalidation Overheads in DSM Systems," *Proc. 19th IEEE Int'l Performance, Computing, and Comm. Conf.,* pp. 597-603, Feb. 2000.

Mainak Chaudhuri received the Bachelor of Technology degree with honors in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, in 1999, and the MS degree in electrical and computer engineering from Cornell University, Ithaca, New York in 2001, where he is currently working toward his PhD degree. His research interests include parallel computer architecture, cache coherence protocol design, and cache-aware parallel algorithms for scientific computation. He also has special interests in theoretical computer science and applied mathematics. He is a student member of the IEEE and the IEEE Computer Society.

Mark Heinrich received the PhD degree in electrical engineering from Stanford University in 1998, the MS degree from Stanford University in 1993, and the BSE degree in electrical engineering and computer science from Duke University in 1991. He is an assistant professor in the School of Electrical and Computer Engineering at Cornell University and a cofounder of its Computer Systems Laboratory. His research interests include novel computer architectures, parallel computer architecture, data-intensive computing, scalable cache coherence protocols, active memory and I/O subsystems, multiprocessor design and simulation methodology, and hardware/software codesign. He is the recipient of a US National Science Foundation CAREER Award supporting novel research in data-intensive computing. He is a member of the IEEE and the IEEE Computer Society.

Chris Holt received the MS degree from Stanford University in 1992, and the BSE degree in computer engineering from Carnegie Mellon University in 1990. He works at Transmeta Corporation. His interests include dynamic binary compilation, computer architecture, garbage collection, and operating systems.

Jaswinder Pal Singh received the PhD degree from Stanford University in 1993, and the BSE degree from Princeton University in 1987. He is an associate professor in the Computer Science Department at Princeton University. His research interests are on the boundary of parallel and distributed applications and multiprocessor systems, both architecture and software, and in applications of high-performance and distributed computing. At Stanford, he participated in the DASH and FLASH multiprocessor projects, leading the applications efforts there. He has led the development and distribution of the SPLASH and SPLASH-2 suites of parallel programs, which are widely used in parallel systems research. At Princeton, he has led the PRISM research group, which does application-driven research in supporting programming and communication models on a variety of communication architectures, as well as in novel applications of high-performance computing such as simulating the immune system. He has coauthored a graduate textbook called "Parallel Computer Architecture: A Hardware-Software approach." He is a Sloan Research Fellow and a recipient of the Presidential Early Career Award for Scientists and Engineers (PECASE). He is a member of the IEEE and the IEEE Computer Society.

Edward Rothberg received the PhD degree in computer science from Stanford University in 1993, the MS degree in computer science from Stanford University in 1989, and the BS degree in mathematical and computational sciences from Stanford University in 1986. He manages the development of the CPLEX mathematical programming package for ILOG, Inc. His interests include linear and integer programming, sparse linear algebra, and parallel numerical computing.

John Hennessy, President of Stanford University, received the BE degree in electrical engineering from Villanova University in 1973. He received the Masters and PhD degrees in computer science from the State University of New York at Stony Brook in 1975 and 1977, respectively. Since September 1977, he has been a faculty member at Stanford University, where he is currently a professor of Electrical Engineering and Computer Science. Prior to becoming President, Professor Hennessy served as the University Provost, the Dean of the School of Engineering, and was chairman of the Computer Science Department. He is the recipient of the 1983 John J. Gallen Memorial Award, awarded by Villanova University to the most outstanding young engineering alumnus. He is the recipient of a 1984 US National Science Foundation Presidential Young Investigator Award and, in 1987, was named the Willard and Inez K. Bell Professor of Electrical Engineering and Computer Science. In 1991, he received the Distinguished Alumnus Award from the State University of New York at Stony Brook. He is a fellow of the IEEE, a member of the National Academy of Sciences, a member of the National Academy of Engineering, a Fellow of the American Academy of Arts and Sciences, and a Fellow of the Association for Computing Machinery. He is the recipient of the 1994 IEEE Piore Award, the 2000 ASEE R. Lamme Medal, the 2000 John Von Neumann Medal, the 2001 Eckert Mauchly Award, and the 2001 Seymour Cray Award. In 2001, he received an honorary doctorate from Villanova, and an honorary degree of science from SUNY Stony Brook.