Application	Remote Memory Bandwidth (bits/cycle/proc)
sieve	9.80
blkmat	1.44
sor	30.20
ugray	6.16
water	3.59
locus	15.07
mp3d	19.91
barnes	3.08

Table 6.2: Average remote memory bandwidth needs of applications under **explicit-switch** multithreading.

Machine (P = 1024)	Bisection Bandwidth (bits/op/proc)
TERA	55.0
CM-5	2.5
DASH	1.8
KSR1	1.6

Table 6.3: Bisection bandwidths of proposed and existing machines if scaled to 1024 processors.

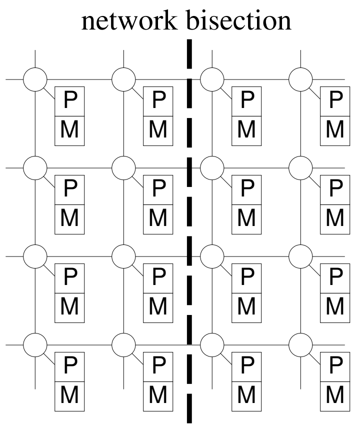

Figure 6.2: A 2-D mesh network and its bisection.

not seem directly comparable to the bisection bandwidths in Table 6.3. The remote memory bandwidth denotes the total network bandwidth used by the applications; whereas bisection bandwidth denotes the bandwidth between two halves of a machines. The amount of bisection bandwidth used depends upon the network and traffic patterns. For example, Figure 6.2 shows a 2-D mesh network with a dashed line drawn across its bisection. If data is laid out so that most traffic is to nearby nodes, then little traffic will cross the bisection. For some applications with regular communication patterns, such as `sor`, good layouts are possible if the network topology matches the communication pattern. However for many applications, the communication patterns and data usage are not predictable, and for some such as `barnes`, there is inherent long distance communication in the algorithm[SHG92].

While good layouts could solve the bandwidth problem, bad layouts could devastate performance by placing several heavily used variables on the same memory module. In this dissertation we do not try to solve the layout problem. Instead, we assume that data is randomly spread across the memory modules in order to avoid bad layouts. With random data layout, half of traffic the will cross the network bisection, and thus a bisection bandwidth of X would be sufficient to support a remote memory bandwidth of $2X$. Furthermore, the comparison between bisection bandwidth and remote memory bandwidth is actually closer than this factor of two because networks do not achieve their peak bandwidth capacity.

Thus comparing the remote memory bandwidth needs of our applications (as high as 30 bits/cycle/proc, from Table 6.2) to the bisection bandwidths of most networks (just 1 or 2 bits/operation/proc, from Table 6.3)[3], we must conclude that these networks will be inadequate for an **explicit-switch** multithreaded system. The only network which can handle these bandwidths is the proposed Tera network, which is in fact a multithreaded system without shared memory caching. This network has perhaps more bandwidth than is actually needed, but was purposefully designed so as not to be bandwidth limited[Smi92]. For the other networks, there must be some mechanism for reducing the bandwidth requirement.

[3]We have expressed our bandwidth results in terms of *bits/cycle/proc* because it is an easily understood unit. However, we have expressed the network bandwidth in the more architecturally independent units of *bits/operation/proc*. These units are equivalent for our simulations because we model a pipelined RISC processor that executes at one operation per cycle.

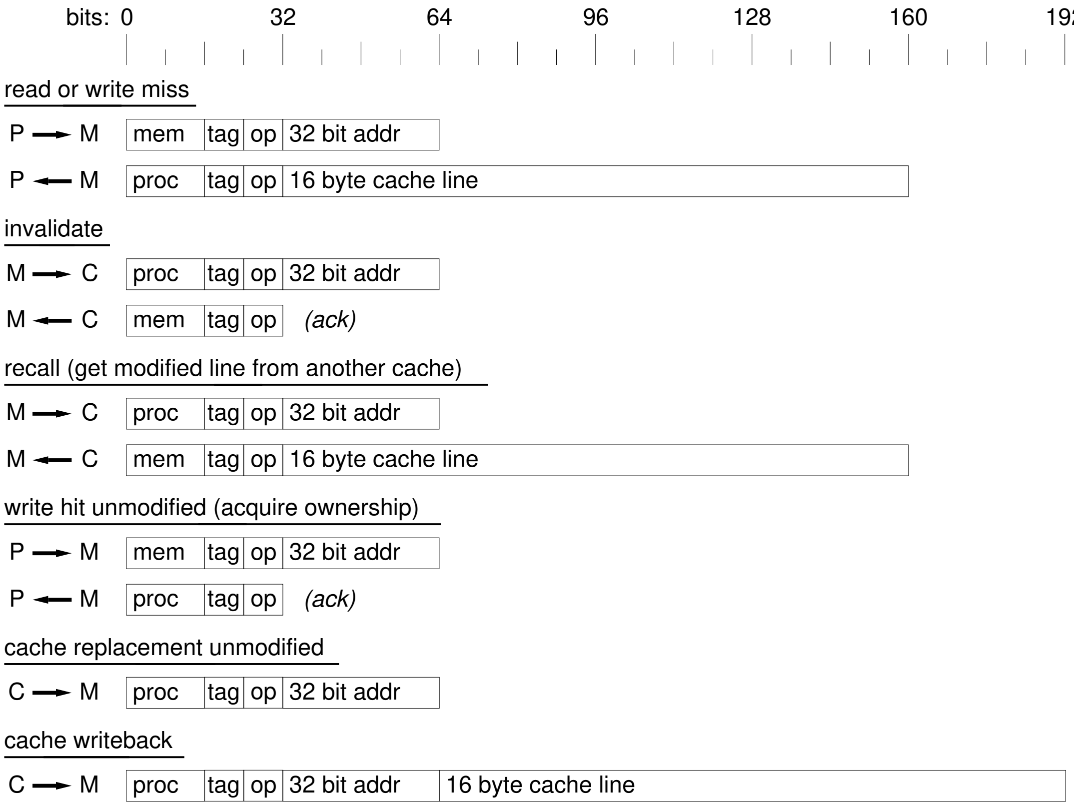

Figure 6.3: Messages used to support coherent caching.

6.1.2 Bandwidth Requirement With Caching

In Chapter 5 most applications showed high cache hit rates and thus caching should be effective at reducing the network bandwidth requirements. The bandwidth reductions, however, will not be as high as the hit rates. Caches typically have large line sizes, and these large lines will require more bandwidth to transmit than the single word memory accesses used on systems without caching. Also, extra traffic will be needed to maintain cache coherency.

Figure 6.3 shows the messages and sizes that we have assumed for our simulations of cache coherent systems. These are similar to the messages for a non-caching system (Figure 6.1), but now the memory returns an entire cache line of data rather than just a single or double word. There are also additional messages, such as invalidation and recall messages, that are used to maintain cache coherency.

Table 6.4 shows the simulation parameters, and Table 6.5 shows the bandwidth

Experiment: **bandwidth under switch-on-miss**			
Application	Processors	Multithreading	• Latency = 200 cycles • Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy • Scheduling = timeout(100) + lock-priority + spin-switch • Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.
sieve	128	1	
blkmat	64	4	
sor	16	4	
ugray	32	4	
water	29	3	
locus	10	3	
mp3d	32	11	
barnes	32	3	

Table 6.4: Experimental parameters for measuring bandwidth under **switch-on-miss**.

Application	Bandwidth (bits/cycle/proc)
sieve	0.10
blkmat	0.91
sor	1.06
ugray	1.09
water	0.50
locus	1.97
mp3d	14.61
barnes	0.18

Table 6.5: Average remote memory bandwidth needs of applications under **switch-on-miss**.

results measured for **switch-on-miss**. The bandwidths are under 2.0 bits/cycle/proc for all of the applications except `mp3d`. This compares to bandwidths without caching (see Table 6.2) of as high as 30 bits/cycle/proc. This large reduction in bandwidth is a clear advantage of systems using caching.

`Mp3d` is able to achieve high execution efficiencies when given enough multithreading, but because it has high cache miss rates, its network bandwidth requirement is an order of magnitude higher than that of the other applications. Such a high bandwidth will probably not be provided unless many applications need it, and thus we must conclude that `mp3d` is incompatible with cache coherent multiprocessors.

6.2 Squeezing Through a Limited Bandwidth Network

So far we have reported just the average remote memory bandwidth needs of the applications. Simple averages, however, give an optimistic view of the demands on the network because actual traffic will be bursty rather than uniformly spread out over time. Applications pass through different computational phases, some of which do more or less computation or communication than others. Furthermore the communication is not spread evenly across the collection of memory modules. Some memory modules are likely to be subject to higher usage than others (a hot spot) because of many parallel accesses to a single shared variable, or simply due to random coincidence.

The network chosen for an actual machine will necessarily be a compromise. It should have enough bandwidth to satisfy the demands of most application most of the time, but will not be able satisfy all the applications all the time. During periods of inadequate bandwidth, the network will limit the performance of the machine to the rate at which messages can squeeze through the network. A properly chosen network should provide sufficient bandwidth that the overall performance loss to periods of inadequate bandwidth is limited.

In the following sections we take a more detailed look at application behavior in order to determine how bursty the traffic is, how much network bandwidth is really needed, and how useful is special hardware for message combining. We develop a performance model for bursty traffic in a bandwidth constrained network, and apply it to the traffic patterns obtained from simulations of our applications.

Certain aspects of traffic patterns, particularly hot spot references to shared vari-

Experiment: bursty traffic under switch-on-miss

Application	Processors	Multithreading
blkmat	256	2
sor	256	4
ugray	256	3
barnes	256	2

- Latency = 200 cycles
- Context switch = 3 cycles if caused by a miss, 0 cycles if caused by the scheduling policy
- Scheduling = timeout(100) + lock-priority + spin-switch
- Each processor has a 64K byte cache with a 16 byte line size and 4 way set associativity.

Table 6.6: Experimental parameters for bursty traffic under **switch-on-miss**.

Application	Problem Size	Simulation Length (cycles)	Processor Utilization	Cache Miss Rate
blkmat	320×320 matrices	1.8 M	93%	35%
sor	768×768 grid	first 20 M	94%	1%
ugray	gears — 160×512 slice of image	first 20 M	88%	12%
barnes	16,384 bodies	first 20 M	72%	15%

Table 6.7: Increased problem sizes of applications.

ables, will be more pronounced in larger systems with more processors. For this reason we have simulated systems as large as possible. The simulations were for 256 processors and are specified in Table 6.6. The problem sizes were increased so as to provide enough work to allow adequate parallelism.

We selected four of the eight benchmarks for the studies in the remainder of this chapter: `blkmat`, `sor`, `ugray`, and `barnes`. We could not use `locus` or `water` because we did not have large enough inputs for these applications. We rejected `mp3d` because it caches poorly and thus is incompatible with a **switch-on-miss** parallel machine, and we rejected `sieve` because its bandwidth usage is so low that it is not interesting for this study.

Table 6.7 lists the increased problem sizes. Unfortunately, these larger problems (except `blkmat`) took far too long to allow executing them to completion. Thus `sor`, `ugray`, and `barnes` were executed only for the first 20 million cycles. We list the processor uti-

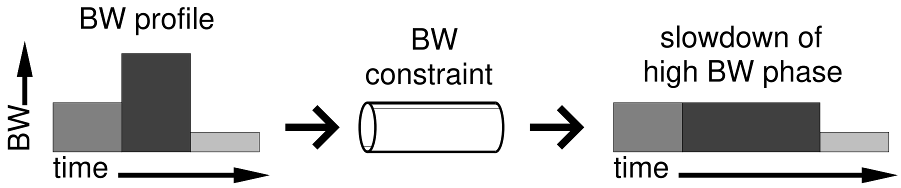

Figure 6.4: High bandwidth phases will slow down as they squeeze through the bandwidth constrained network.

lizations over this period since the execution efficiencies can not be computed unless the applications are run to completion. For `ugray` and `barnes` the cache miss rates have increased because of the larger problem sizes.

6.2.1 Remote Memory Bandwidth

In this section we look at how the total remote memory bandwidth needs of the applications vary over time. We will predict the performance that will be achieved on machines with limited bandwidth networks by developing a performance model and then applying it to the bandwidth needs of the applications.

Squeeze Performance Model

Figure 6.4 shows the basic idea of our performance model. We start with a bandwidth profile of an application. This is obtained from simulations, and it shows the varying bandwidth needs of the application as a function of time. For most applications the bandwidth will not be uniform. Instead, the applications will have different phases with different bandwidth needs as shown in the figure. The network of a real machine will have a maximum bandwidth capacity, which is represented in the figure by the pipe labeled *BW constraint*. For our performance model, we assume that during phases when an application needs less bandwidth than is available, it will execute at full speed. But during phases when then bandwidth needs exceed the network bandwidth capacity, we assume that execution slows down and makes progress at the rate as which messages squeeze through the network.

Figure 6.5 formally specifies our performance model. This model is much more accurate than simply looking at the average bandwidth over the entire run of the execution, but it is still optimistic. Under some adverse traffic patterns there may be some links of the network or memory modules that are more heavily used than others. These will be

Squeeze Performance Model

$$\text{slowdown} = \frac{\sum_{i=1}^{n} t_i \max\left(1, \frac{\text{bw}_i}{\text{bw}_{\text{net}}}\right)}{\sum_{i=1}^{n} t_i}$$

n = number of phases
t_i = duration of phase i
bw_i = bandwidth needed in phase i
bw_{net} = bandwidth available

Figure 6.5: Performance model of an application having phases with varying bandwidth needs being executed on a machine with limited network bandwidth.

bottlenecks and could further slow down the execution of the machine. Hopefully, such bottlenecks will be rare when data is spread randomly across the machine as we have assumed.

Simulation Results

In practice, applications do not exhibit long uniform phases as suggested by the squeeze performance model. The processors are all semi-independent systems which issue occasional messages into the network. Together they form a very bursty system. At some particular point in time, there might be a large burst of new messages resulting from random coincidence. However, this burst will not slow down the machine if on subsequent cycles there is a compensating lull in new traffic. On a small time scale the network and its buffering serve to smooth out the traffic.

To take into account this natural smoothing of the traffic, we have gathered our simulation data over intervals of 100 cycles. Much shorter sample intervals would be pessimistic since they would report bursts of traffic that could be smoothed out by a real network, and likewise much longer sample intervals would be optimistic since they would smooth over long bursts of traffic that on a real network *would* have a performance impact. We chose the value of 100 cycles because it is half of the expected 200 cycle network latency. This latency will consist of both physical delays and congestion delays with perhaps half of the latency due to each. The congestion delay is caused by the jostling of messages as they move through the network, and this is the period in which small bursts of traffic are smoothed out.

Figure 6.6 shows snippets from the remote memory bandwidth profiles of the applications. These snippets are for 100,000 cycles, whereas the applications were simulated for 20,000,000 cycles. The vertical bars in these graphs each represent a sample interval of 100 cycles. During each interval, the sizes of all messages sent into the network were added together to give a single sample value: the remote memory bandwidth total for that interval. These sample values were then normalized to bits/cycle/proc.

These bandwidth profiles graphically show both the short term and long term burstiness of traffic, but the entire 20,000,000 cycle profiles are too large to be included in their entirety. We therefore present the complete bandwidth profiles by sorting the sample intervals. The sorted profiles are shown in Figure 6.7. Sorting the samples allows drawing a compact graph that more clearly shows the fraction of time the applications operate at various bandwidth levels. For example, when running `sor` the network sits idle (or nearly idle) for 54% of the time. For 12% of the time (the interval of sorted samples from 54% to 66%) the bandwidth is between 0 and 1 bits/cycle/proc, and the rest of the time it is higher. About 10% of the time it is higher than 4 bits/cycle/proc. The other applications exhibit less variance in their bandwidth profiles.

These sorted profiles can be used to easily visualize and compute the performance loss that will result when the applications are run on a machine with a bandwidth constrained network. For `blkmat`, for example, if the network has a bandwidth capacity of 1 bit/cycle/proc, 85% of the program execution has bandwidth needs less than this, and will be unimpeded, but the remaining portion will be slowed down, much of it by about a factor of four.

Table 6.8 shows the precise slowdowns of the applications under various bandwidth limits. These were calculated by applying the squeeze performance model to the bandwidth profiles. `Blkmat`, for instance, under a bandwidth of 1 bit/cycle/proc will slow down by a factor of 1.37.

Using this table, one can choose an appropriate bandwidth level such that the

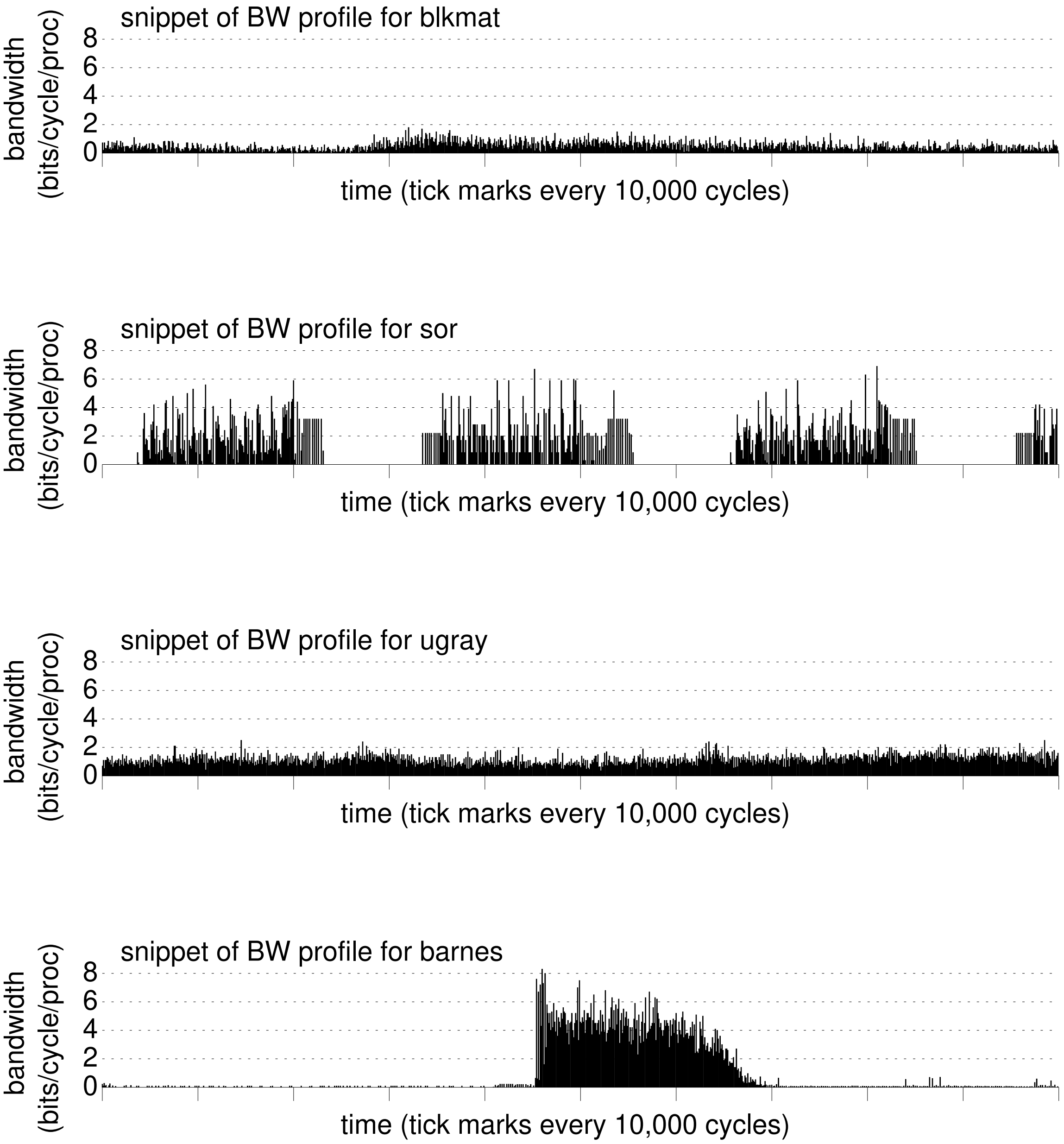

Figure 6.6: Snippets of remote memory bandwidth profiles.

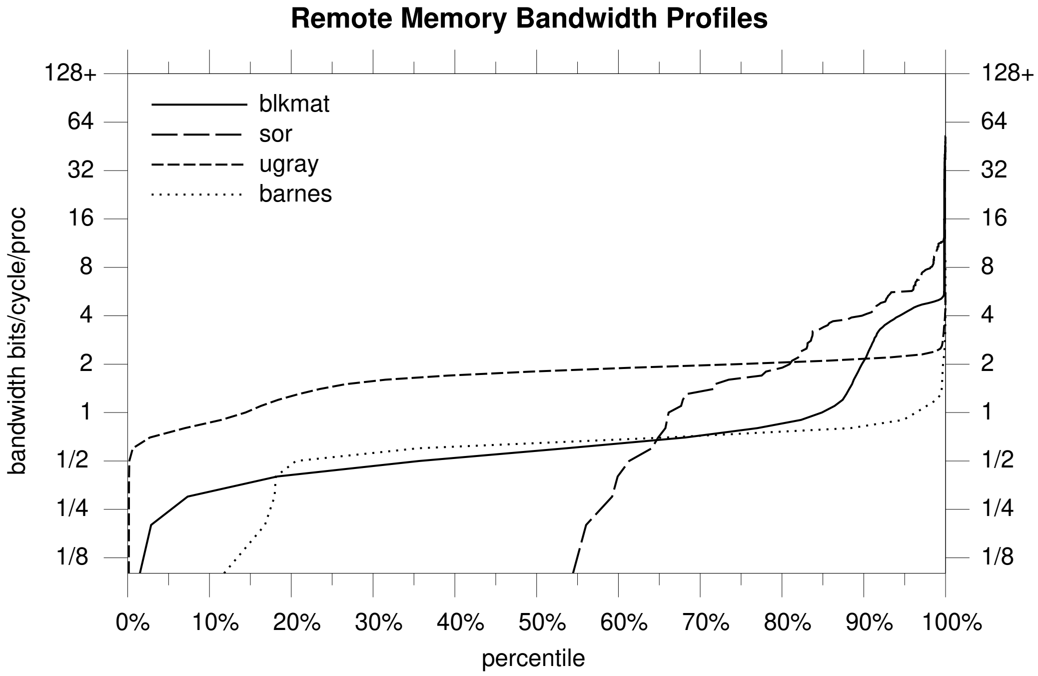

Figure 6.7: Sorted profiles of the applications' remote memory bandwidth usage.

cost of increasing the network bandwidth is warranted by the increase in performance. Clearly it is not worth doubling the size and cost of the network just for a 1% or 2% performance increase. Thus the proper remote memory bandwidth of the network is from 2 to 4 bits/cycle/proc.

6.2.2 Hot Spot Memory Modules

The next part of the bandwidth picture concerns how evenly references are spread across the memory modules. If a particular memory module is getting more traffic than other modules, it is dubbed a *hot spot*. A hot spot become a problem when traffic directed to it exceeds its capacity. At this point messages to it will be delayed and will back up into the network.

Not only will messages to the hot spot be delayed, but as congestion backs up into the nodes of the network, messages destined for other memories can get blocked as well. The congestion quickly compounds and can spread across the network in a process called

bw	blkmat	sor	ugray	barnes
32				
16				
8	1.01	1.01		
4	1.02	1.07		
2	1.13	1.29	1.02	
1	1.37	1.85	1.71	1.01

Table 6.8: Slowdown factors under various bandwidth limits.

tree saturation[PN85].

Hot spots occur for several reasons. We classify hot spots into three categories which we call *location*, *layout*, and *random*. Location hot spots occur when there are many accesses to a particular shared variable, such as a shared counter. They are called location hot spots because they involve a single location. Layout hot spots occur when data is spread across the machine in a regular fashion, but because of the particular access patterns of the applications, some memory modules are much more highly utilized than others. This is analogous to the stride problem for vector computers, and can be diminished by randomly spreading addresses across the memory modules. We have done this randomization of addresses in our simulations to avoid layout hot spots, but this accentuates the third category of hot spots which are due simply to random variation. At various points in time, simply through random chance, some memory modules will be more heavily utilized than others.

A Pessimistic Performance Model

For a long lasting hot spot, it is clear that because of tree saturation the performance of the entire machine will be slowed down to match the service rate of the hot spot memory. However a short lived hot spot may only slow down those processors directly involved, or possibly because of synchronization and data dependencies, the delays may propagate to other processors as well.

Our model takes a pessimistic outlook and assumes that hot spots will slow down the entire machine. During each sample interval, the most heavily accessed memory module thus determines the machine slowdown for that interval. The total slowdown is calculated by applying the squeeze performance model to the links and traffic directly entering and exiting the hot spot memory module.

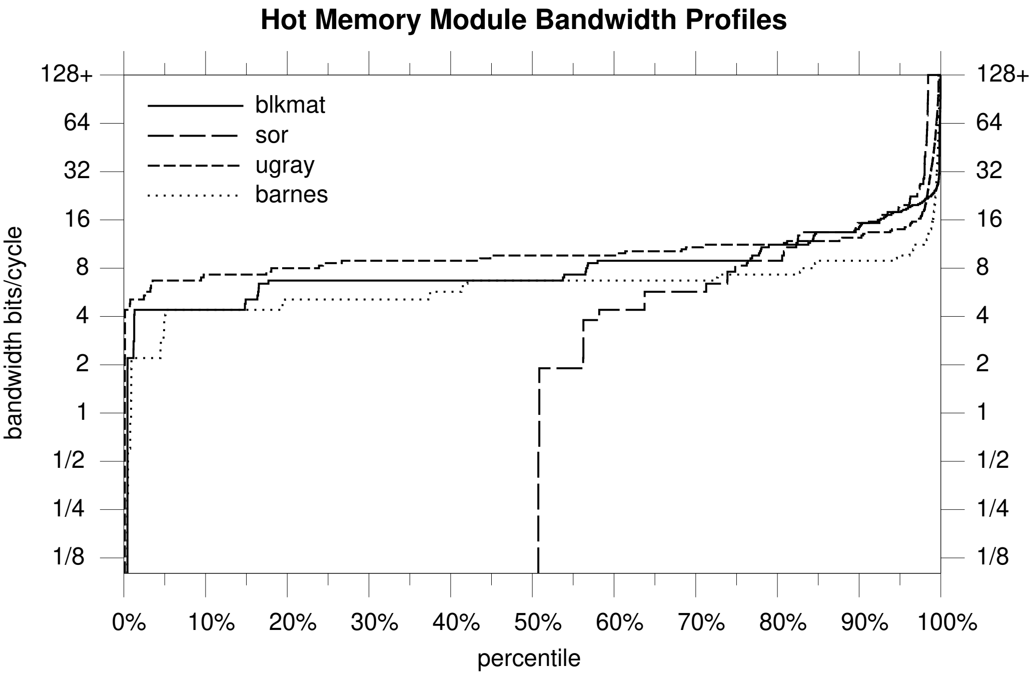

Figure 6.8: Sorted profiles of the applications' hot memory module bandwidth usage.

Simulation Results

Figure 6.8 shows the simulation results for the hot spot memory module bandwidth. These are typically a factor of 4 to 8 larger than the remote memory bandwidths, and thus hot spots are an important part of the traffic picture.

Table 6.9 shows the application slowdowns computed using our performance model. Based on these results we can say that the network needs a memory module bandwidth of

bw	blkmat	sor	ugray	barnes
32	1.01	1.08	1.02	1.01
16	1.03	1.19	1.05	1.03
8	1.25	1.55	1.36	1.10
4	2.23	2.41	2.65	1.77
2	4.44	4.26	5.31	3.49
1	8.88	8.02	10.61	6.97

Table 6.9: Slowdowns factors based on hot spot memory module bandwidth.

		Network		Memory Module	
Application	Average BW	BW	over design factor	BW	over design factor
blkmat	0.98	4	4	16	16
sor	1.24	4	3	32	26
ugray	1.68	2	1.2	16	10
barnes	0.59	1	1.7	8	14

Table 6.10: Over design factors for the network and memory modules.

16 bits/cycle. We should qualify this by restating that this is a pessimistic model, and that it assumes either rapid onset of tree saturation or propagation of delays to processors not directly involved in the hot spot. We performed additional simulations with a sample interval of 500 cycles in order to gauge how sensitive our results are to this assumption. These simulation, which are probably optimistic, suggest that a memory bandwidth of 8 bits/cycle is adequate. Thus our conclusion is that memory module bandwidth should be from 8 to 16 bits/cycle.

Compared to our results in Section 6.2.1 indicating a remote memory bandwidth of 2 to 4 bits/cycle/proc, the memory module bandwidth is a factor of 4 higher. The direct implication is that networks having higher local bandwidths than bisection bandwidths are advantageous. For instance the fat tree network in the CM-5[LAD+92] was designed so that the lowest level of the network has four times the bandwidth of the upper levels. Another example is the networks of M. T. Raghunath[RR93] that provide higher local bandwidths as a means of getting high utilization of the bisection bandwidth. A third example is mesh networks that allows adaptive routing of traffic around the hot spot memories.

Another implication of the higher memory module bandwidths is that the memory modules must be over designed so that they have far more capacity than will be needed on average. We can calculate this over design factor by dividing our performance model's bandwidth suggestions by the average bandwidths actually used by the applications. Table 6.10 shows this calculation (for each application individually) for both the network and the memory modules. The network and memory module bandwidths used in this table were taken from Tables 6.8 & 6.9 at levels that allowed achieving slowdowns ≤ 1.10. The network over design factors are moderate and range from 1.2 to 4. The memory module over design factors are much larger and range from 10 to 26.

Such large over design factors are required to service the hot spots in the memory access patterns. These hot spots arise because of the inevitable non-uniformity of the random message distribution. An analogous problem is the random distribution of n balls into n buckets. On average each bucket will receive 1 ball, but the worst case bucket will receive $\Omega(\log n / \log\log n)$ balls.

6.2.3 Location Hot Spots

The elimination and reduction of location hot spots has been the subject of a large body of research[DK92, GGK+82, MCS91, PN85, Ran89, YTL86]. These involve either hardware combining of messages or software combining trees. Hardware support is typically for fetch-and-add operations, from which many highly parallel synchronization techniques can be built[GGK+82]. Software techniques have been devised for barriers and synchronous reductions[MCS91, YTL86].

Despite the large amount of research on combining techniques, there has been little previous work done on measuring how beneficial combining would be for real applications. This is partly a chicken and egg problem because without hardware support, programmers have little incentive to use fetch-and-add like operations. Although it has been suggested that fetch-and-add is useful even if not combined[MCS91] because it is a simple atomic operation that can be performed quickly at the memories. We have provided such a non-combining fetch-and-add operation in our simulation system, but we have used it only a few times.

In general, ordinary memory requests, such as several reads to a single location, can also be combined. In this section we determine an upper bound on the benefits of hardware combining. Our simulations measure (indirectly) the total number of accesses to each individual memory location during a sample interval. We then use these numbers as our upper bound on combining. In other words, we assume that all references to a single location during a sample interval can be combined. This is optimistic for two reasons. First, combining of different reference types (such as a read, a fetch-and-add, and a write) is very complex and unlikely to ever be implemented. And second, our sample interval of 100 cycles is long enough that in a real network messages will often pass through the routing nodes before their potential combining partners arrive.

Figure 6.9 shows the amount of traffic at the hottest (most heavily used) location

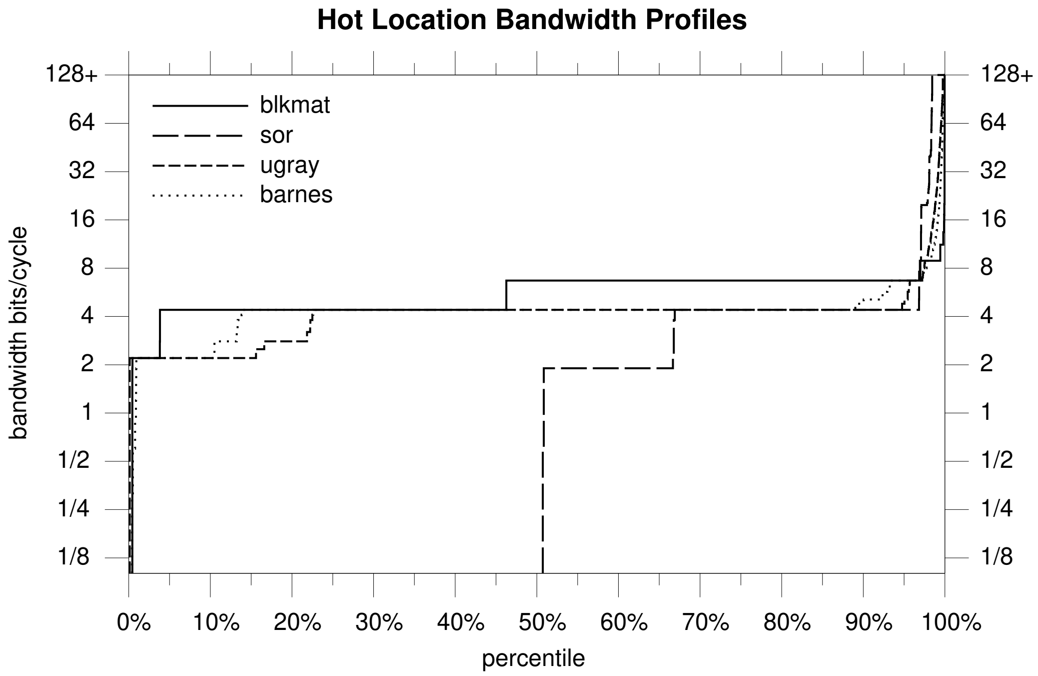

Figure 6.9: Sorted profiles of the applications' hot location bandwidth usage.

during each sample interval. As before, the samples are sorted from lowest to highest. To interpret this data we need to know the correspondence between bandwidth and the actual number of memory operations that occurred. Figure 6.3 showed our assumptions for message sizes. The messages needed to service a simple read miss constitute a total of 224 bits (7 words) of traffic. A read miss is the most common operation in the network and we will use it as our basis for calculation. When normalized to bits/cycle over a 100 cycle sample interval, as we have used, a single read miss uses a bandwidth of 2.24 bits/cycle.

Using `ugray` as an example, the lowest 16% of the intervals show a hot location bandwidth of 2.24 bits/cycle, which is equivalent to the traffic from one read miss message. This means that no location was referenced more than once during these intervals. Next there are some sample intervals that have slightly higher bandwidths. These most likely represent a single access that caused some invalidation traffic. After these, the rest of the intervals all the way up to the 94th percentile show a bandwidth of 4.48 bits/cycle, which equals 2 messages. These two messages might be combined, but such limited combining has little benefit and is not the motivation for combining hardware.

Chances for combining large numbers of messages are rare. For example intervals with 10 or more messages directed to a single location occur during only 2% of the intervals for `sor`, 1.1% of the intervals for `ugray`, and 0.6% of the intervals for `barnes`. For `sor` these hot spots are primarily due to barriers and could be eliminated with either a special purpose barrier network as on the CM-5[LAD+92] or perhaps with software barriers[MCS91, YTL86]. Since serious location hot spots are so infrequent, we do not believe that hardware combining is justified.

We should note that two of the applications were modified after the initial run of these simulations. `Sor` showed an unexpected hot spot which was caused by all processors resetting a shared flag when only one processor needed to reset it. This was easily amended.

For `ugray`, there was a problem with contention for a single lock that guarded a free list. This lock was a bottleneck, but had not been noticed before because `ugray` had never previously been run with so many processors. The bottleneck was eliminated by using a parallel free list.

6.3 Summary and Implications

For most applications, caching is effective at reducing the network bandwidth requirement. Thus despite the cost and complexity of providing coherent caches, caching is likely to be cost effective because of the reduction in network cost afforded by allowing a skinnier network than on systems without caching.

When burstiness is taken into account, our simulation results and performance model have shown that the networks for large shared memory cache coherent multiprocessors should provide a remote memory bandwidth of from 2 to 4 bits/cycle/proc and a memory module bandwidth of from 8 to 16 bits/cycle. The higher bandwidth requirement at the memory modules is needed to accommodate random hot spot traffic.

While these numbers have been normalized by the processor cycle time and therefore should be widely applicable, an adjustment is needed to account for differences in processors. Specifically, our results are based on the MIPS R3000[Kan89], a pipelined RISC processor operating at one operation per cycle. A superscalar processor operating at two operations per cycle, for example, would thus need to support twice the network bandwidth.

Network	Routing Nodes	Links[4] per Node	Link Width if constrained by: Remote Memory Bandwidth	Link Width if constrained by: Memory Module Bandwidth	Total Wires
Ring	256	4	$\boxed{128}$	4	65536
2-D Torus	256	8	$\boxed{8}$	2	8192
3-D Torus	256	12	$\boxed{4}$	1.3	6144
Butterfly[5]	1024	8/2	2	$\boxed{8}$	36864
Fat-Tree[6]	120	24/4	$\boxed{8}$	4	15360
Hypercube	256	16	$\boxed{2}$	1	4096

Table 6.11: Example networks and the link widths they need to provide 4 bits/cycle/proc remote memory bandwidth or 16 bits/cycle/proc memory module bandwidth for a 256 processor machine. The larger link width is selected in order to satisfy both bandwidth requirements.

Table 6.11 lists an assortment of different network topologies and some key parameters. The link widths are based on the bandwidth values suggested by this research. They were calculated so that the networks would provide either a remote memory bandwidth of 4 bits/proc/cycle or a memory module bandwidth of 16 bits/cycle. The larger of these two link widths was then selected and used for calculating the total wire count, which is a rough estimate of network cost. The key observation to be made is that all of the networks except for the butterfly provide more local bandwidth than bisection bandwidth, and this extra local bandwidth is beneficial because it provides the higher bandwidth needed by the hot spot memory modules.

[4] All of these network calculations are based on using pairs of unidirectional links to provide bidirectional routing. For the butterfly and fat-tree networks, two numbers are given for links per node. The first value applies to the routing nodes and the second value applies to the processor and/or memory nodes.

[5] This butterfly network calculation is based on 2 by 2 switches.

[6] This fat-tree calculation is based on 4 by 4 switches and on decreasing by a factor of 2 the total number of links at each higher level in the tree.

Chapter 7

Miscellaneous Studies

This chapter is a collection of miscellaneous studies and experiments that were not critical for the main presentation, but nevertheless are important details and issues that arose during this research.

7.1 Synchronization

The synchronization methods of current shared memory multiprocessors use spinning when it becomes necessary to wait for a synchronizing event to occur[GT90, MCS91]. For a single threaded processor spinning is acceptable because the processor does not have anything else it could do while waiting. For a multithreaded processor, however, spinning is wasteful because the processor could instead be gainfully executing its other threads. In this section we first review spin based synchronization techniques and then present on-spinning alternatives.

7.1.1 Spin Waiting

Atomic Bus Transactions

On bus based shared memory multiprocessors, synchronization is facilitated by an atomic memory instruction such as: `test-and-set`, `swap`, or `compare-and-swap`. These instruction are called atomic because they allow reading and then writing a memory location in an indivisible fashion so that there are no intervening memory accesses.

These atomic instructions are then used to build all higher level synchronization primatives. For example, a lock on the Sequent Symmetry is just a memory location and its value[GT90]. If the value is 0, the lock is free; if it is 1, the lock is taken. To obtain the lock a processor executes a `swap(addr,1)` instruction. This instruction atomically reads the old value at the address and writes the new value of 1. If the `swap` instruction returns an old value of 0, the lock has been obtained, but if it returns the value 1, then the lock is already taken. In this case, the processor spins: continuously reading the memory location until it changes to 0, at which point the processor retries the `swap` instruction.

When the lock is free, this is very efficient since there is just a single memory reference used to obtain the lock, and later just a single `write(addr,0)` instruction to release the lock. The problem arises when the lock is simultaneously desired by several processors. One processor will obtain the lock; the others will incessantly read the lock location waiting for its release. These reads would saturate the bus to shared memory except that most are filtered out by the caches. After the first read, the value is cached and subsequent reads simply spin on the value in the cache. When eventually the processor holding the lock relinquishes it by writing a 0 to the lock location, it triggers invalidations (or updates) of all the caches. At this point the remaining contenders all race to obtain the lock.

With n processors contending for the lock, each of these races involves $O(n)$ bus accesses. By the time a group of n processors all get their turn with the lock, there will have been $0(n^2)$ bus accesses to the lock location. There are a number of more sophisticated lock implementation which try to reduce this traffic[And90, GT90, MCS91]. The best of these reduce bus traffic to $0(n)$ by building a software queue in which the waiting processors can all spin on local flags. Releasing a lock involves clearing the flag belonging to the next processor in line.

All of these lock implementations use spinning. The difference between them is simply how much of the traffic is local or global.

Fetch-And-Add

The NYU Ultracomputer project[GGK+82] proposed an innovative synchronization instruction: `fetch-and-add` (`f&add`). `F&add(addr, value)` reads the specified memory address, returns the contents as the operation's result, and then adds the specified value and stores the sum back into the addressed location.

A simple use of `f&add` is to have each of n processors execute `f&add(`X`, 1)`. If initially $X = 0$, the values returned will all be unique and go from 0 to $n-1$. These values might then be used to select unique tasks for each processor in a parallel computation.

The power of `f&add` comes from the fact that multiple `f&add` messages can be combined in a tree like fashion as they proceed through a butterfly network. If combining works well, a group of simultaneous `f&add`'s will get fully combined so that only a single message actually reaches the memory module. The result message returned from the memory module will then be split apart and the correct return values computed as the message(s) return back down the combining tree. This can be done in such a way that the responses are the same as if the `f&add`'s had been performed sequentially. Combining allows congestion at the memories due to hot spots to be eliminated (although our results in Section 6.2.3 suggest that such congestion is rare in our applications).

Gottlieb[GGK+82] shows that data structures such as parallel queues can be designed using `f&add` and combining so that there is no serial bottleneck. In other words, hundreds of processor can simultaneously insert and remove entries from the queue without ever entering a critical section (where only a single processor has exclusive access to the internal queue data structures).

Although `f&add` based synchronization routines (along with combining) can eliminate memory hot spots (if they were a problem), these routines still use spinning in order to wait for a synchronizing event: such as the release of a lock or the insertion of an entry in a queue. Thus from the point of view of a multithreaded processor, cycles would still be wasted by threads waiting on synchronization operations.

Full/Empty Bits

Synchronization on the HEP[Kow85] was done through full/empty bits associated with each memory location. For example, there was a write instruction that would set the full bit when it wrote a value, and there was a read instruction that would check that the full bit was set before reading the value. If the location was empty, the reading thread would wait until an appropriate write occurred. These full/empty bits allowed very fast and fine-grained synchronization.

Since HEP was a multithreaded processor, spinning the processor while waiting would be wasteful. Instead there was a separate unit called the Storage Function Unit that did the spinning.

7.1.2 Non-Spinning Synchronization

All of the spin based synchronization techniques reviewed in the previous section treat synchronization as simply a “fancy” memory reference: a memory reference that also does a test or addition. The desired synchronization constructs are then built by using these “fancy” memory references in conjunction with spinning,

In fact, synchronization is really something more than this. Obtaining a lock is really a request to be notified when the lock becomes available, and releasing a lock should instigate the notification of a waiting thread. Likewise, checking in at a barrier is a request to be notified when all cohorts have also checked in, and the final thread to check in should instigate this notification.

Fundamentally, synchronization mechanisms such as locks and barriers involve a message from a thread to a synchronization agent(Sync-Agent) and then a subsequent message from the Sync-Agent to the appropriate synchronizing thread. For example, a lock is requested by sending a message to the Sync-Agent. If the lock is available, the Sync-Agent immediately sends a reply message granting the lock to the thread and marks the lock as taken. If another request for the lock then arrives while the lock is taken, the Sync-Agent queues this request, and later when the lock is eventually released, the Sync-Agent sends a message granting the lock to the first queued requester. This message based synchronization provides a mechanism for eliminating spinning. Queue-On-Sync-Bit[GVW89] and Queue-On-Lock-Bit[Gus92] are similar mechanisms for building queues of waiting processors, but they were designed without consideration of multithreading and thus use local spinning.

In this section we present a collection of synchronization operations which can be built upon this understanding of synchronization as message passing and therefore directly implementing it as such. In fact, in a large shared memory multiprocessors, memory references are also messages as well: a message is sent to a memory modules to read a value, and a reply message is returned to deliver the result. Because everything is actually implemented with messages, our message based synchronization will turn out to have a simple implementation that fits naturally into the design of our multithreaded processor.

Messages

What was described above as the synchronization agent, is really just the interface to a memory module. For normal memory operations the interface receives messages, does the memory operation, and sends replies. It also must send and receive any messages needed to maintain cache coherency. For synchronization operations, the memory interface sends the messages dictated by those operations.

A synchronization variable is a memory location just like any other variable, the only difference is that it is accessed via synchronization instructions rather than normal memory access instructions.

We have provided the following synchronization operations in our simulator:

lock: A message is sent to request a lock, and when the lock becomes available, a message is returned granting the lock. An unlock message is used to release a lock.

barrier: Each thread sends a message when it is ready to check in at a barrier, and when all threads have checked in, barrier completion messages are returned to each of the threads[1].

fetch-and-add: This is the same as for the NYU Ultracomputer except that there is no combining. A `f&add` message is sent to the memory, the addition is performed there, and the reply message returns the fetched value.

wait: This is similar to the full/empty bits on the HEP except that it is just a synchronization: there is no associated data transfer. A message is sent to wait for a specified flag to be set. If the flag is already set, a completion messages is immediately returned, otherwise the response occurs later when the flag is set. There are also messages for setting and resetting the flag.

The messages formats and their sizes are shown in Figure 7.1. These messages are the same format as the messages used for regular memory operations, except that no data is associated with most of the synchronization operations. They are therefore compact messages; most are just one or two words long.

[1] In our general simulations we have not modeled the limited network bandwidth and the serialization at the memory module of the many barrier messages. In Section 6.2 we addressed the network bandwidth limits and found that barriers are infrequent enough in our applications that the occasional congestion they cause has only a minor performance impact. Later in this section we will mention some congestion free alternatives.

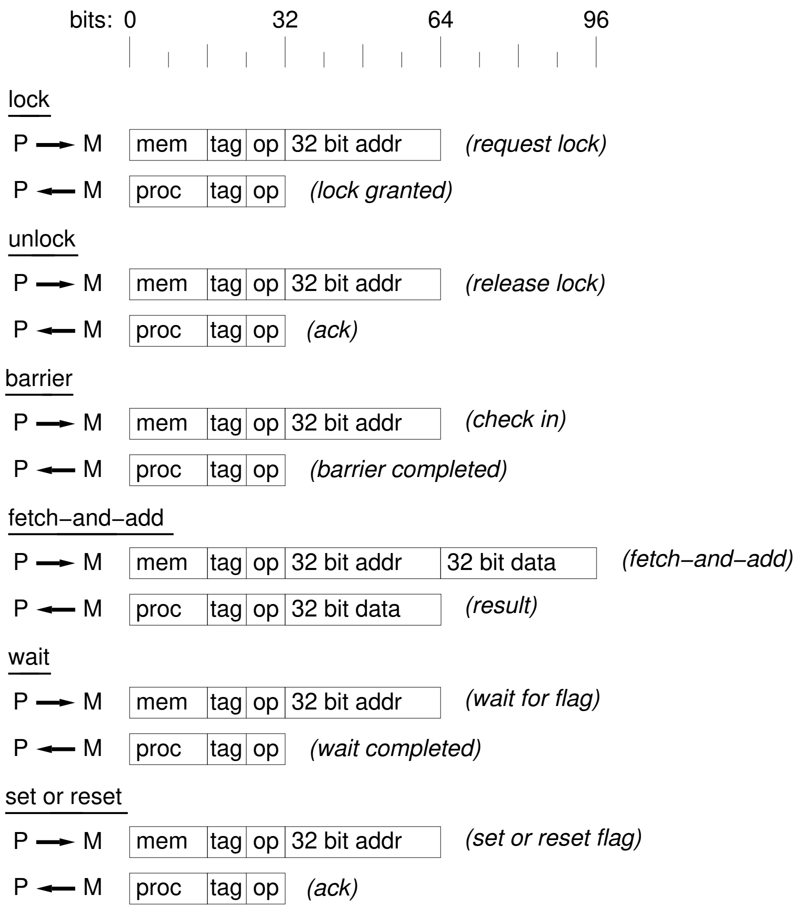

Figure 7.1: Messages for synchronization operations.

Application	cycles/sync/processor			
	lock	barrier	`f&add`	wait
sieve	—	240K	480K	—
blkmat	17K	1.7M	—	—
sor	—	24K	—	—
ugray	115K	—	8K	—
water	5K	2.9M	—	—
locus	19K	78M	—	—
mp3d	—	117K	7K	—
barnes	41K	1.7M	340K	176K

Table 7.1: Frequency of use of various synchronization operations. These are expressed in terms of the average number of cycles between synchronization operations of a particular type, on a per processor basis. K = one thousand cycles. M = one million cycles.

Frequency of Use

Table 7.1 shows the frequency of use of these synchronization operations under **switch-on-miss**[2]. Locks and barriers were used by most of the applications, `f&add`'s by some, and waits were used only by `barnes`. In fact, the original codes for `mp3d` and `barnes` did not explicitly use `f&add` or wait operations. We inserted these operations when we replaced spin based equivalents.

Spinning was eliminated from most of the applications, but `ugray` and `locus` still have spinning in their parallel task queues. Most of the spinning occurs for these applications as the computation is completing and threads are waiting for any new tasks to be made available. Because of this spinning, it was useful to have **spin-switch** as part of the scheduling policy (see Section 5.2.1).

All of these applications are coarse grained in their use of synchronization. The heaviest synchronization uses are: locks in `water` are obtained at rates of once every 5,000 cycles, barriers in `sor` are used once every 24,000 cycles, and `f&add`'s in `mp3d` are used once every 7,000 cycles. Since the use of `f&add`'s (and/or any other type of synchronization) is infrequent, this corroborates our results in Section 6.2.3 that suggested that message combining in the network would provide little benefit.

[2]The experimental parameters are the same as in Table 6.4.

Implementation

Our message based synchronization fits nicely with the design of a multithreaded processor. When a thread issues a normal memory access into the network, that thread is then context switched out and it waits for the response message. The exact same things occur for synchronization. For example a thread requests a lock by sending a message into the network, and then it waits for the response message saying that the lock has been granted. The only difference between the two cases, as far as the processor is concerned, is that memory references have fairly uniform latencies (a few hundred cycles), but locks have varied latencies (depending upon how many other threads are also waiting for the lock and how long they each hold it).

Because issuing a memory reference or a synchronization reference both cause descheduling of a thread, and their responses both cause its reactivation, both types of references are treated identically by the processor. However because synchronization references may have long latencies, strictly round robin scheduling is inappropriate. The processor should not wait on a specific thread if other threads are ready to execute. First-in-first-out (FIFO) is a simple scheduling policy, and the policies looked at in Section 5.2.1 were all just slight modifications to FIFO.

At the memory modules, memory references and synchronization references *do* have to be treated differently. There will be some sort of memory interface unit whose job is to extract messages from the network, process them, and send the replies. For normal memory references, this processing will be simple: read the requested memory block (line) and send it. However for some memory references, there will be coherency operations, such as invalidations, that also must be performed. These will involve the sending and receiving of additional messages. For synchronization operations, the memory interface unit must check if the synchronization event has occurred and queue waiting requests.

The main complexity in implementing these synchronization operations is maintaining the queue of waiting processes. Figure 7.2 shows one possible implementation[3]. A lock variable is just the address of a small structure in shared memory. The first location contains a flag which says if the lock is free or busy (taken). The next two locations contain pointers to the head and tail of a list of waiting threads. This three word lock structure should reside on a single memory module. Also each memory module will need a local array

[3]The acknowledgement responses to unlock messages were not shown in this figure in order to make it simpler.

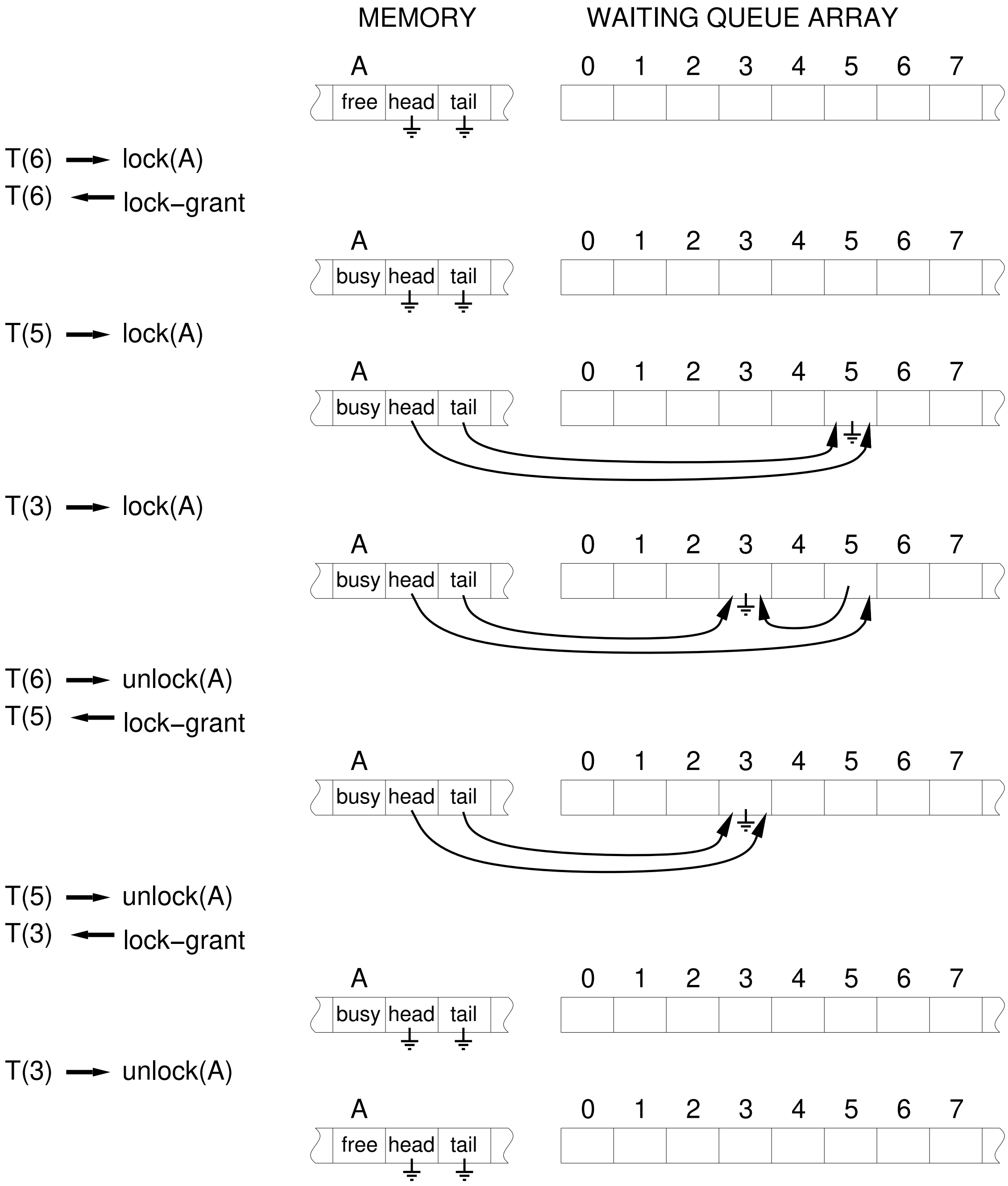

Figure 7.2: Operation of the waiting queue for a lock.

in which it can build waiting queues as linked lists. This array contains one location for each thread, but only one such array will be needed per memory module, regardless of the number of locks.

In the example in Figure 7.2, A is a lock variable which is initially set to free. Thread 6 is the first to request the lock and is therefore immediately granted the lock. Threads 5 and 3 then request the lock and are queued up in FIFO order. Next thread 6 releases the lock, which causes the lock to be granted to the thread at the head of the waiting list: thread 5. When thread 5 releases the lock, the lock is granted to thread 3. And finally when thread 3 releases the lock, the lock returns to the free state.

Since the three word lock structure and the waiting queue array all reside on the same memory module, they can easily be updated atomically. The memory interface unit simply performs all its operations for the lock variable before servicing the next incoming message.

If an application has multiple lock variables, they should be spread across the memory modules to avoid unnecessary hot spots. Since the number of lock variables is unlimited, there may be several lock variables on each memory module. These can all share a single waiting queue array since the threads in each waiting queue will be distinct. This is true because it is only possible for a thread to be waiting on one lock at a time, and thus it could never be on more than one queue. It is perfectly valid, however, for a thread to obtain nested locks. They just must be obtained one at a time.

Because of the complexity of synchronization (and cache coherency) it is likely that the memory interface will be some sort of programmable device. In fact, each memory module will likely be connected to one of the processors, and the synchronization and cache coherency might be handled by a quick interrupt to the processor. The Wisconsin Wind Tunnel[HLRW92] uses a CM-5 in this fashion; the processor manages the cache coherency protocol. We suggest letting the processor handle synchronization operations in the spirit of active messages[vECGS92]. This allows the synchronization operations to be changed and supplemented rather than being designed into the machine.

Besides locks, other synchronization operations such as barriers, waits, and queues can also be built with messages so that no spinning is required. In addition to eliminating spinning, performing complex synchronization operations at the memory modules has the advantage of being faster than building them out of several simpler synchronization operations that each involve a network traversal delay[BR90].

If contention is a problem, many synchronization operations can be implemented in a distributed fashion. Barriers can be implemented with software combining trees[YTL86] or a potentially faster technique called the dissemination barrier[MCS91]. Index distribution and work queues can be implemented using the low contention techniques of Herlihy[HLS92]. These are based on counting networks and do not use spinning if there is an atomic memory operation such as fetch-and-add.

7.2 Line Size for Minimizing Bandwidth

In this section we study the affect of cache lines sizes on the network bandwidth needs of our applications. A large cache line has the potential of decreasing network bandwidth because the headers for routing and specifying a memory access are of fixed size (see Figure 6.3), but the data payload varies with the cache line size. With a larger line, the fraction of bandwidth used for data is higher.

In practice, a larger line size might might actually increase the bandwidth requirement for several reasons. First, the requesting processor may not use all of the locations in a cache line. These unused locations use bandwidth when they are brought across the network, but do not otherwise affect performance. Second, a large line size increases the likely hood of false sharing. This is the case where two different processors access different parts of a single cache line, and the cache line is ping-ponged back and forth between the processors even though they are not actually sharing any variables. Third, larger line sizes imply fewer total lines in the cache and thus increase the probability that useful data will get replaced with unwanted data (cache pollution).

Figure 7.3 shows the bandwidth usage of the applications when run with cache line sizes ranging from 8 to 128 bytes. The experimental parameters are shown in Table 7.2. These experiments did not use multithreading. However, we expect that the same relative relationship between bandwidth and line size will continue to hold for multithreaded systems.

Most of the applications have increasing bandwidth needs with larger line sizes. The best choices for minimizing bandwidth usage are either 8, 16, or 32 byte cache lines, depending upon the application. A 16 byte line size is the best overall choice, and that is the value we have used throughout the studies in this thesis.

O'Krafka[O'K92] also looked at traffic as a function of cache line size. He studied

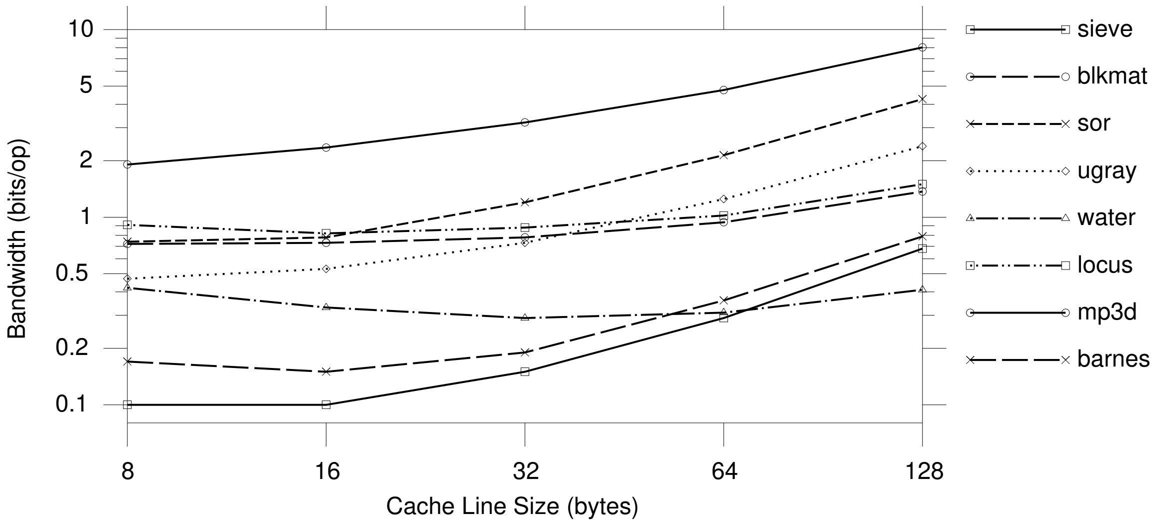

Figure 7.3: Bandwidth as a function of line size.

Experiment: Bandwidth versus Line Size			
Application	Processors	Multithreading	• Latency = 200 cycles • Each processor has a 64K byte cache with 4 way set associativity. The line size was a parameter of the experiment.
sieve	128	1	
blkmat	64	1	
sor	16	1	
ugray	32	1	
water	29	1	
locus	10	1	
mp3d	32	1	
barnes	32	1	

Table 7.2: Experimental parameters for measurements of bandwidth versus line size.

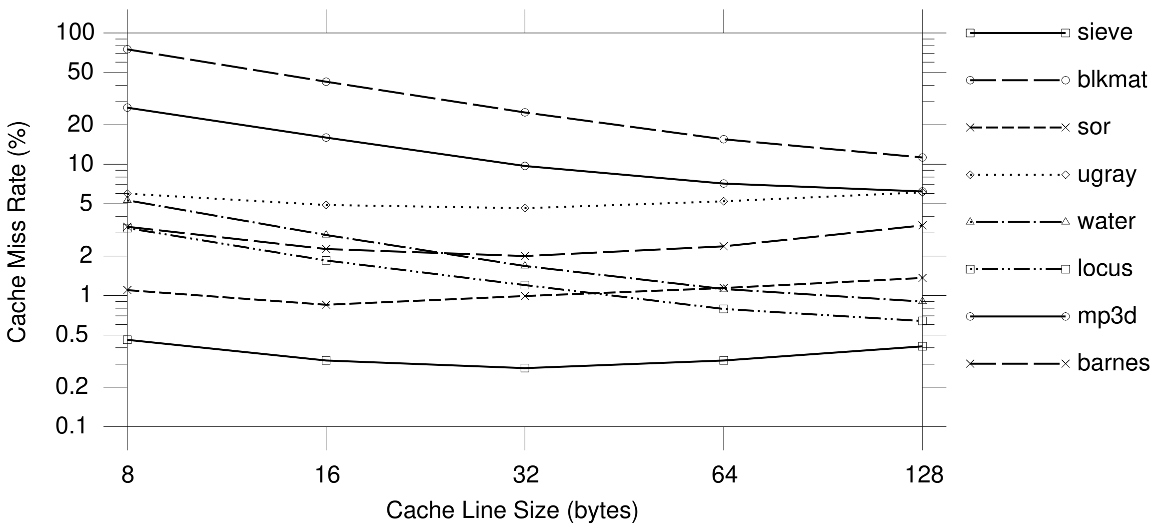

Figure 7.4: Miss rates as a function of line size.

three application: `verf`, `ugray`, and `locus`, and three different lines size: 4 bytes, 16 bytes, and 64 bytes. Traffic was minimized for `verf` with a 4 byte line, for `ugray` with a 16 byte line, and `locus` with a 64 byte line. He also compared invalidation based protocols with update based and competitive protocols. He found that update and competitive protocols "are only appropriate for multiprocessors with 100 or fewer processors", and we therefore limited our studies to invalidation based protocols.

The rising bandwidth usage with large line sizes suggests that the problems of unused data, false sharing, and cache pollution do occur for many of the applications. Figure 7.4 shows the miss rates as a function of line size. These would be expected to decrease with larger line size, and do for some of the applications (`blkmat`, `mp3d`, `water`, and `locus`). The diminishing rates of decrease with larger line sizes suggest some of the data fetched in large lines is unused. The other applications (`ugray`, `barnes`, `sor`, and `sieve`) shows increasing miss rates with large line sizes, which suggest false sharing and cache pollution are occurring for these applications.

For those configurations showing decreasing miss rates, the performance should be higher since fewer stalls on remote references are occurring. Figure 7.5 shows the processor utilizations as a function of line size and confirms this inverse relationship between miss rates and performance. However, since miss rates do not diminish with increased line size

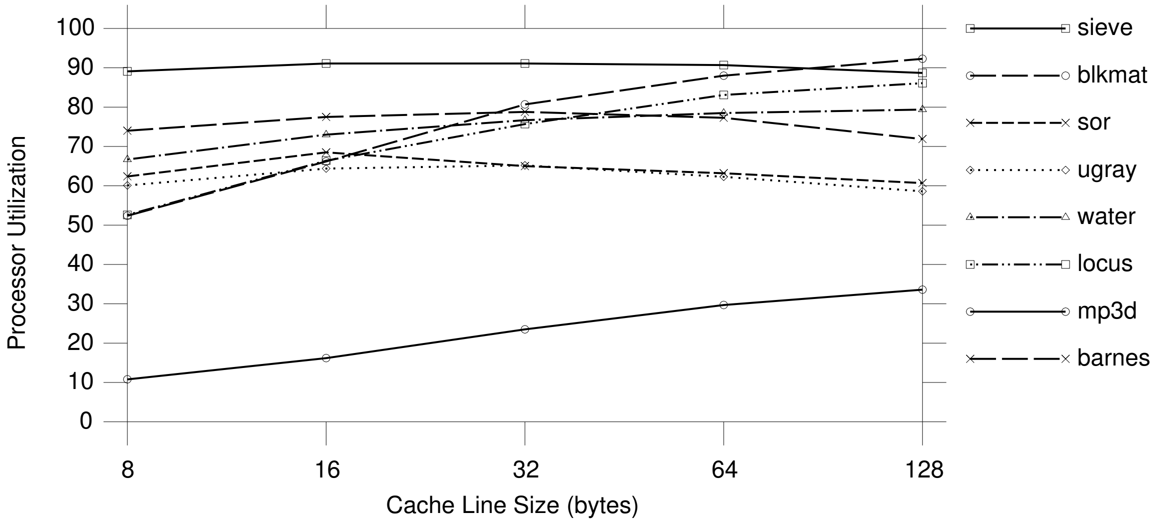

Figure 7.5: Processor utilization as a function of line size.

for all of the applications, performance increases for some applications but declines for others.

The goal of this section is to determine an appropriate line size for minimizing the network bandwidth requirement. Simply looking at the bandwidth as a function of line size can be misleading because bandwidth usage might be low simply because the processors were idle most of the time. Figure 7.6 adjusts for this possibility by normalizing the bandwidth results from Figure 7.3 by the processor utilization results from Figure 7.5. (This sort of normalization actually occurs when multithreading is used to increase performance to a higher level.) The earlier conclusion that a 16 bytes line size is the best overall choice for minimizing bandwidth still holds, although several applications now have slightly lower bandwidth requirements with a 32 byte line size.

Network bandwidth is not the sole issue in choosing a line size. Another major factor is the overhead in terms of cache tag space and directory storage that is needed to locate lines and manage coherency. The number of lines is directly proportional to the amount of this storage, and thus doubling the lines size halves the amount of tag and directory storage. Because of the large amount of directory storage, this may be a more important factor than bandwidth. The KSR1[Ken92], for example, has chosen a 128 byte line size in order to limit the size of its distributed directories.

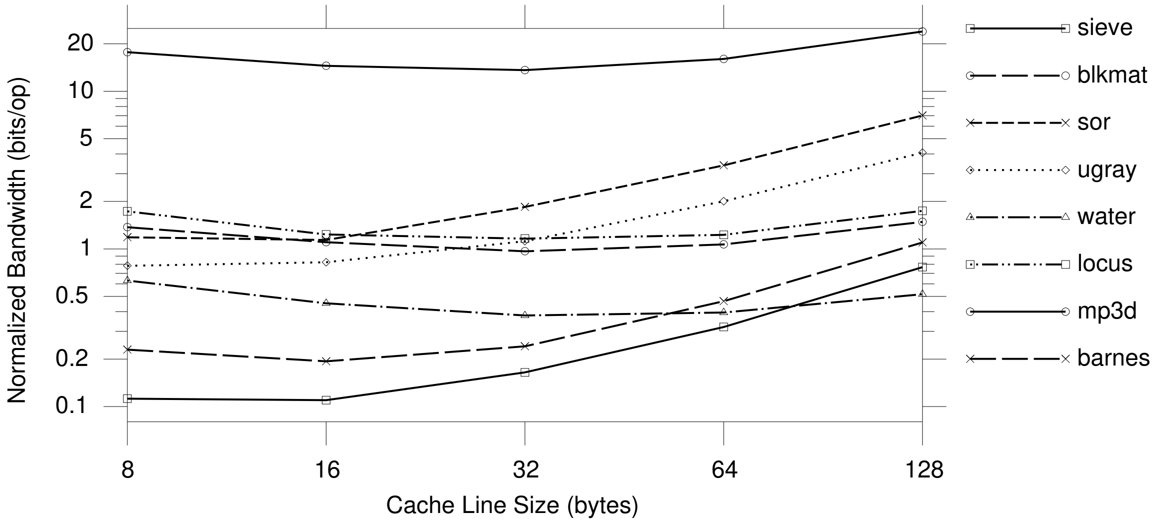

Figure 7.6: Bandwidth (normalized to 100% processor utilization) as a function of line size.

7.3 Cache Degradation due to Multithreading

One concern with multithreaded processors is that they may have poor cache behavior if the threads on a processor compete for space in the cache and knock out data used by each other[ALKK90, CGL92, SBCvE90, WG89]. The interaction between threads as they access the cache can be destructive or constructive. The destructive case occurs when a thread knocks out data that *would have been used* by other threads. The constructive case occurs when one thread brings in data that *is latter used* by other threads.

We think of the processor as conceptually having three caches: one for instructions, one for local data, and one for shared data. Throughout this research we have only considered the shared data cache. We expect that the instruction cache should have constructive interaction between the threads since they are all from a single SPMD (Single Program Multiple Data) program. For many of the applications the threads are synchronized so that they execute in same region of code simultaneously. For local data, the cache interactions will be destructive, and caches should clearly be larger in order to accommodate the multiple threads. Figure 7.7 shows the cache miss rates for the shared data cache as a function of multithreading[4].

[4]The simulation parameters are the same as in Table 5.6.

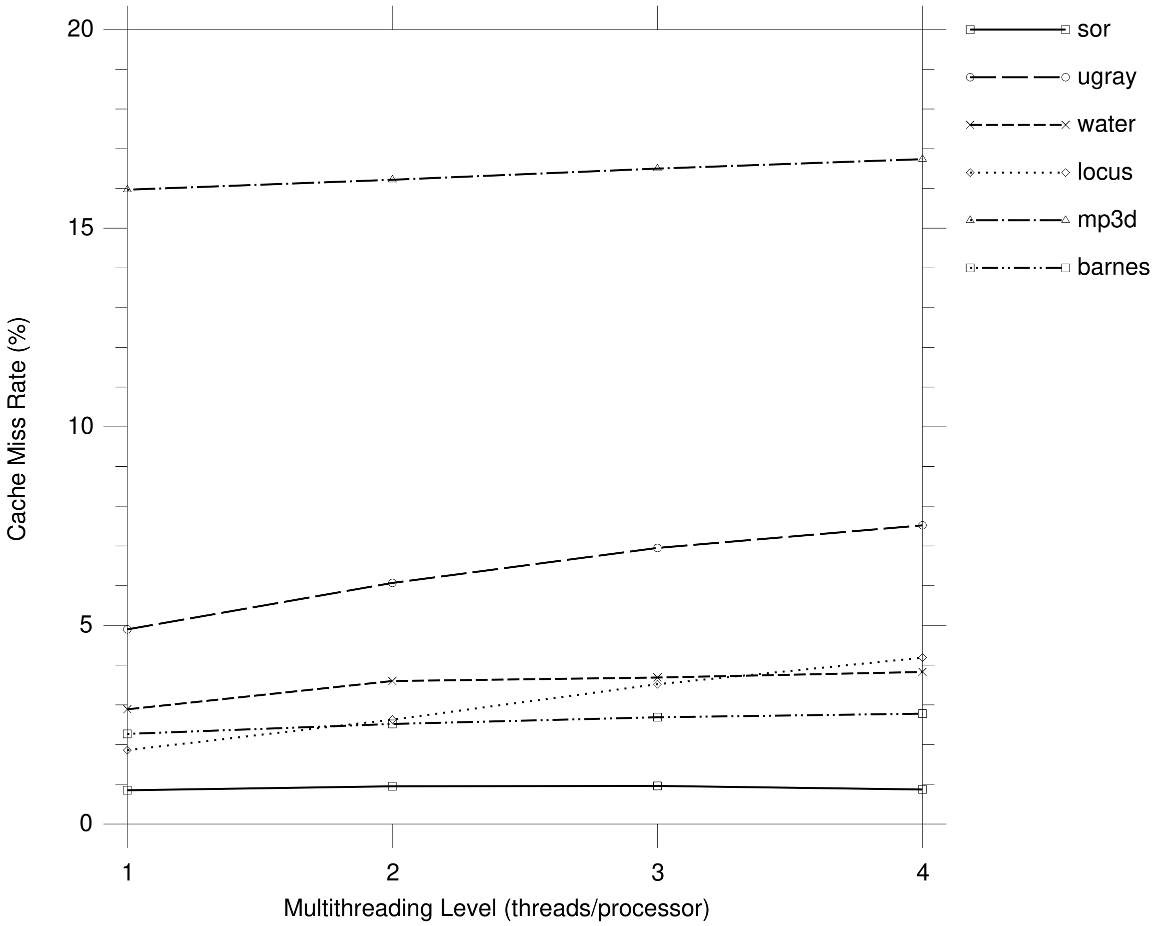

Figure 7.7: Cache miss rates as a function of multithreading.

Generally the miss rates increase slightly with multithreading. Saavedra-Barrera [SBCvE90] modeled this increase in miss rates based on the assumption that with multithreading of N, the cache would behave as if it were partitioned into N subsets, each of size $1/N$th of the original cache. Then based on an analytic model of cache behavior, he derived an expression for the miss ratio as a function of multithreading: $m(N) = m(1)N^K$, where $m(N)$ is the miss ratio with multithreading N, and K is a constant that depends on the behavior of the applications. This model does fit the behavior of many of the applications, but the values of K vary widely.

A major factor in this variation in cache behavior is whether the threads interact constructively or destructively. For some applications this depends on the order in which

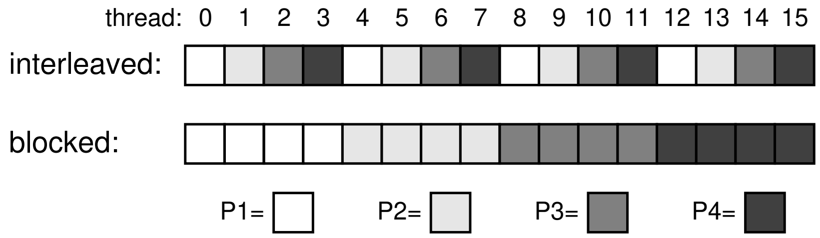

Figure 7.8: Assignment of threads to processors.

threads are assigned to processors. For example, the `sor` application works on a two dimensional array that is partitioned into rectangular blocks with one block per thread (see Section 2.2.3). Interaction between threads occurs for the data along the edges of these blocks, and if abutting blocks (threads) are assigned to the same processor, the common edges will interact constructively in the cache.

Figure 7.8 shows two possible orders for assigning threads to processors. The blocked order is better for `sor`, `barnes`, and `blkmat` because they partition the problem by thread id numbers; and thus threads working on neighboring regions are assigned to the same processor. Blocked ordering is actually worse for `water` because the particles are not isotropically distributed; and thus blocked assignment aggravates load imbalance on the processors. For the other applications, work scheduling is less structured and the thread ordering is unimportant. For the simulations in this thesis, we used blocked assignment for all of the applications except for `water`, for which we used interleaved assignment.

7.4 Longer Latencies

The final issue which we address in this chapter is what happens when latencies are longer than 200 cycles. We should expect that *without* multithreading the increased latencies will mean longer waits for remote references and thus lower efficiencies. *With* multithreading, we should expect that more threads will be needed to hide the longer latencies.

Figures 7.9 & 7.10 show simulation results for the applications at latencies of 200, 500, and 1000 cycles[5]. As expected, single-threaded efficiencies drop with increased

[5]The simulation parameters are the same as in Table 5.6 except that the latency has been varied and higher multithreading levels were used for some applications.

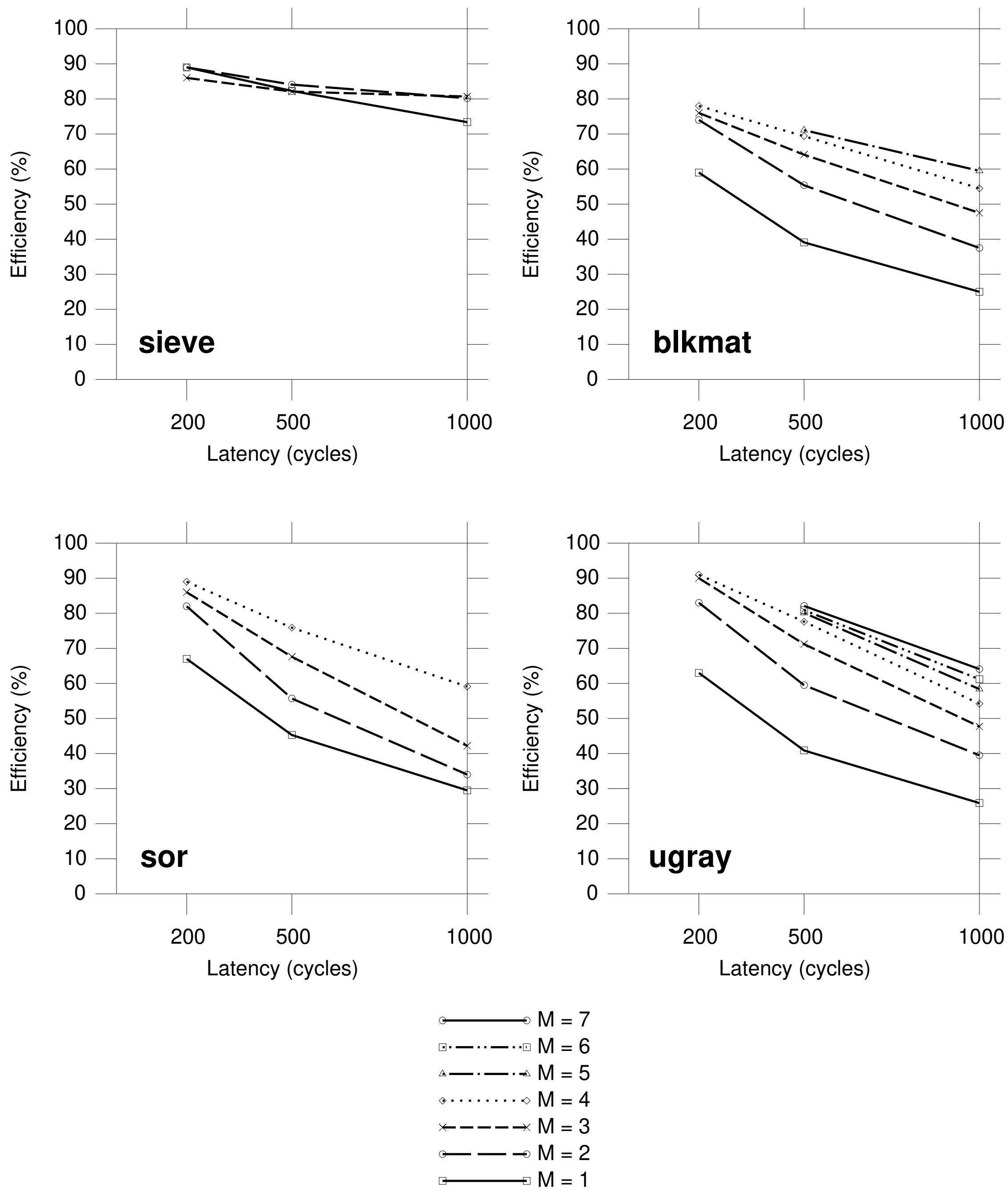

Figure 7.9: Efficiencies with longer latency.

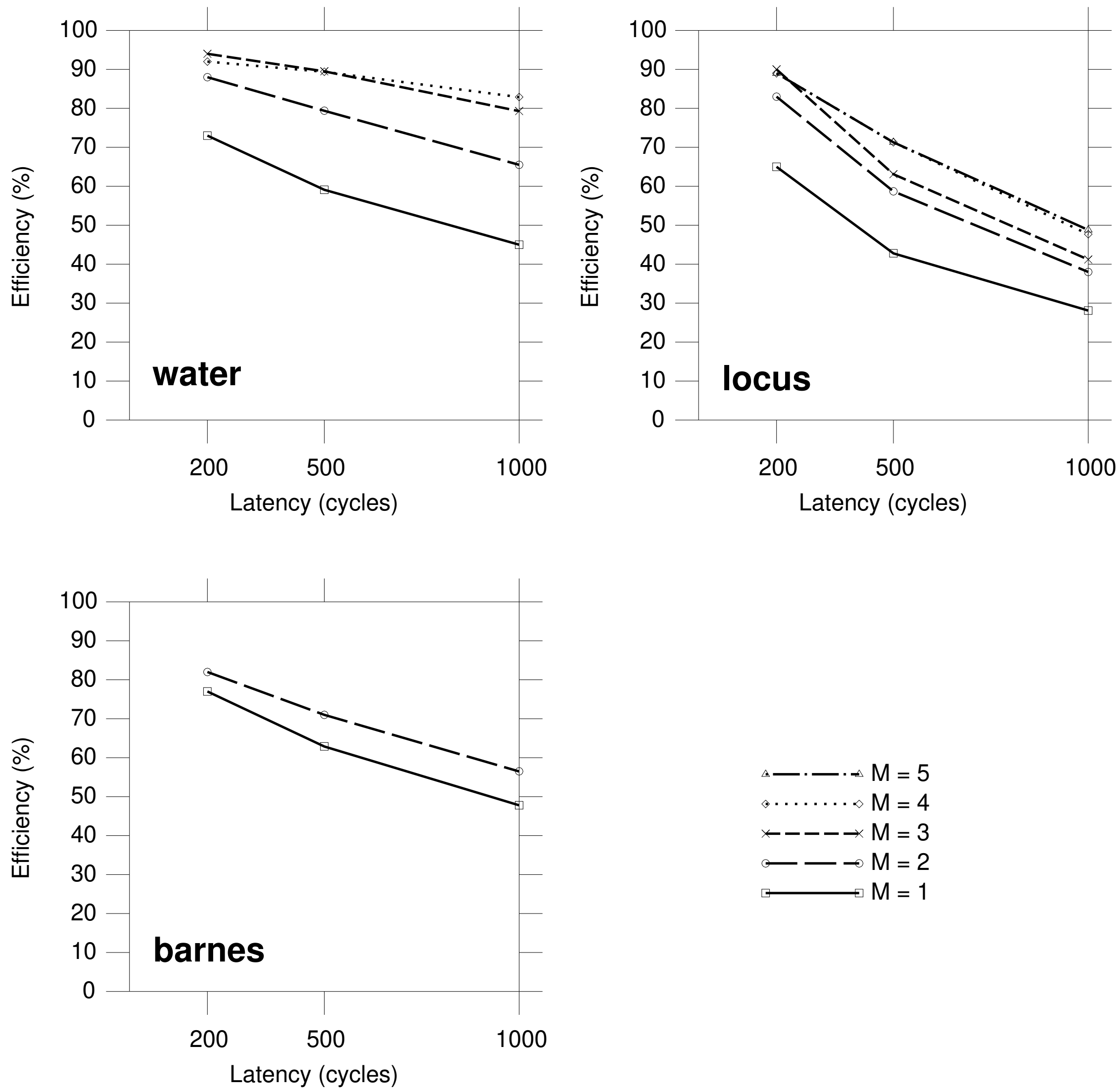

Figure 7.10: Efficiencies with longer latency.

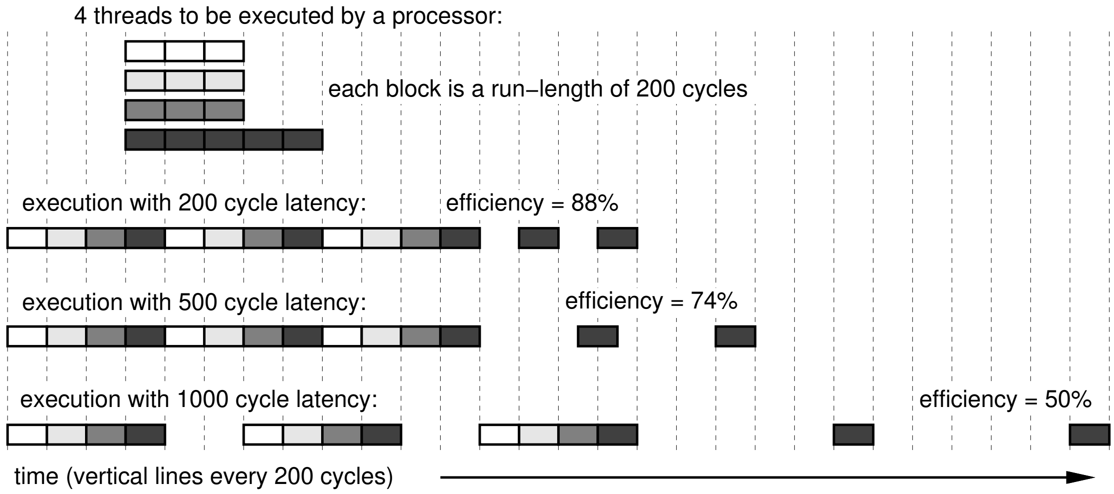

Figure 7.11: Load imbalance between threads has an increasing performance impact with longer latencies.

latencies. Multithreading, however, is not able to bring efficiencies back up to the same levels achieved under smaller latencies. Thus multithreading is unable to completely hide the affects of increased memory latency.

There are a number of factors in this decrease in attainable performance under longer latencies:

Short Run-Lengths: When several threads have coinciding short run-lengths (and thus the latency is not hidden) the stalls last longer because the latency is longer.

Load Imbalance: When threads finish a phase of computation (reach a barrier) at different times, the last thread to finish takes a longer time to finish because of the longer waits for each of its memory references. This is shown in Figure 7.11, where four thread are shown. The first three thread each execute for three run-lengths of 200 cycles, but the fourth thread executes for five run-lengths of 200 cycles. As the network latency increases, completion of the unbalanced thread takes a larger fraction of the total execution time.

Lock Contention: A typical critical section involves: obtaining a lock, doing a few operations and references, and then releasing the lock. When latencies are longer it takes longer for an individual thread to complete a critical section and thus locks are held for longer periods of time, increasing lock contention.

Finer Granularity: As more threads are demanded from a fixed sized problem, the problem must be divided into smaller pieces or work. This finer partitioning leads to more communication and thus shorter run-lengths.

There also several ways to ameliorate the impact of longer latencies:

Larger Problem Sizes: When the applications are run with larger problem sizes, the granularity of tasks can be increased and/or more threads will be available to hide latency.

Larger Caches: Larger caches can decrease the miss rates, which both increases average run-lengths and decreases the number of points at which stalling might occur.

Better Load Balancing: More careful load balancing of threads will help limit the duration of the slow completion process of the last thread.

Despite the lower efficiencies achieved with multithreading under very long latencies, multithreading provides larger net performance gains. The typical application (such as `sor`) has a 100% performance improvement (from 30% to 60% efficiency) under multithreading with a 1000 cycle latency. Whereas with a 200 cycle latency, typical performance improvements were 33% (from 60% efficiency to 80% efficiency).

Chapter 8

Hardware Support

Most previous research into multithreading machines has involved complex hardware to support the switch-every-cycle model[ACC+90, HF88, Kow85, PC90]. These machine were built to switch every cycle so as to allow a fast pipeline implementation without concern for data dependencies. In addition, this multithreading mechanism is used to tolerate long unpredictable memory latencies. These machines typically also provide sophisticated synchronization (using full/empty bits on memory), and support powerful programming models allowing rapid and dynamic creation of threads.

Unfortunately, every hardware capability has its cost. TERA[ACC+90], for instance, allows fast dynamic creation of threads, and it provides 128 banks of 32 registers to hold these threads' register values. This huge amount of hardware complicates the machine because it slows down the access time to the register file. Alternatively, Monsoon[PC90] limits the size of the register file by severely restricting the number and lifetimes of registers, which undoubtably has a negative performance impact. It remains unclear if these machines offer advantages in either performance or ease of programming compared to the simple multithreaded shared memory model studied in this dissertation.

We have studied multithreading models which we feel have a balance between computational flexibility and implementation simplicity. Only a small number of threads are allowed per processor and thus the register file can be kept reasonably small. The programming model involves a static set of threads that is used for the lifetime of the program and thus support for fast dynamic thread creation is not needed. The synchronization mechanisms are simple and similar to remote memory references. Finally, thread scheduling uses simple policies and only switches threads at special events. In this chapter we will present