Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory

Todd C. Mowry
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA 15213
tcm@cs.cmu.edu

Charles Q. C. Chan and Adley K. W. Lo
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{charlesc,load}@eecg.toronto.edu

Abstract

A key challenge in achieving high performance on software DSMs is overcoming their relatively large communication latencies. In this paper, we consider two techniques which address this problem: prefetching *and* multithreading. *While previous studies have examined each of these techniques in isolation, this paper is the first to evaluate both techniques using a consistent hardware platform and set of applications, thereby allowing direct comparisons. In addition, this is the first study to consider* combining *prefetching and multithreading in a software DSM. We performed our experiments on real hardware using a full implementation of both techniques. Our experimental results demonstrate that both prefetching and multithreading result in significant performance improvements when applied individually. In addition, we observe that prefetching and multithreading can potentially complement each other by using prefetching to hide memory latency and multithreading to hide synchronization latency.*

1. Introduction

There has been considerable interest recently in exploiting collections of workstations or PCs connected by commodity networks as less expensive alternatives to tightly-coupled multiprocessors. To help simplify the task of writing parallel applications, software can provide a shared memory abstraction across the machines with the help of the virtual memory system [17]. For certain classes of applications, these software distributed shared memory (DSM) systems can deliver performance which is comparable to hardware cache-coherent machines of a similar scale [6, 11]. However, for applications with larger communication demands, the performance can be disappointing.

[0] To appear in *HPCA-4*, February 1-4, 1998.

A key stumbling block to achieving higher performance on software DSMs is the relatively large communication latency. In contrast with tightly-coupled multiprocessors, where remote miss latencies are on the order of half a microsecond [16], the remote miss latencies for software DSM on moderately aggressive hardware are closer to half a *millisecond* [7]—i.e. roughly three orders of magnitude slower. This large communication latency affects not only remote memory accesses, but also synchronization operations. Since communication latency is already known to be a significant bottleneck even in tightly-coupled multiprocessors, it is reasonable to expect the much larger latencies in software DSMs to make the problem even worse.

1.1. Software DSM Performance

To illustrate the impact of communication latency on software DSM performance, we ran a collection of applications taken primarily from the SPLASH-2 suite [26][1] using TreadMarks [12] (a state-of-the-art software DSM implementation) on eight 133 MHz IBM RS/6000 workstations connected by a 155 Mbps FORE Systems ATM LAN. (Further details of the hardware platform are given later in Section 2.2). Figure 1 shows a detailed breakdown of the resulting execution times, as measured on the real hardware using the high resolution timers under AIX 4.1. The normalized execution times are broken down into the following four categories, from top to bottom: time spent stalled waiting for (i) synchronization and (ii) remote memory misses, respectively; (iii) time spent executing DSM system software (e.g., the memory coherence protocol); and (iv) time spent doing useful computation.[2]

[1] The exception is SOR, which is taken from the TreadMarks distribution.

[2] The "*Busy*" time also includes interrupt times associated with software DSM, since we cannot isolate them otherwise.

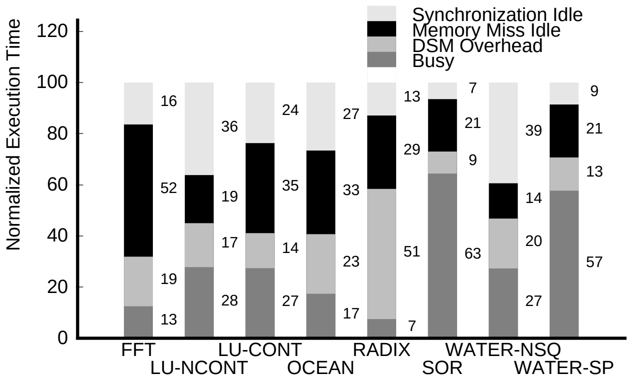

Figure 1. Execution time breakdown using TreadMarks on eight ATM-connected workstations.

As we see in Figure 1, six of the eight applications spend over half of their time stalled waiting for either remote memory references or synchronization. Therefore techniques for coping with the large communication latencies are clearly important.

1.2. Latency Tolerance Techniques

The first step in dealing with latency is to exploit caching to avoid communication. Software DSMs accomplish this by using local memory as a cache for remote locations. The second step is to buffer and pipeline remote accesses through a relaxed memory consistency model, which TreadMarks does using *lazy release consistency* [13]. TreadMarks also uses a *multiple-writer protocol* to avoid the effects of false sharing. The net effect of all of these optimizations is to reduce the amount of communication so that it more closely approximates the inherent communication due to true data sharing and synchronization. While these techniques go a long way toward improving performance, the remaining latency is still painful (as we saw in Figure 1), and hence we would like to cope with it as well.

To tolerate the latency of reading remote memory, we must separate the *request* for data from the *use* of that data, while finding enough useful parallelism to keep the processor busy in between. The two main techniques for accomplishing this are *prefetching* [4, 23] and *multithreading* [2, 9, 14, 15]; the distinction between the two is that prefetching finds the parallelism within a *single thread* of execution, while multithreading exploits parallelism across *multiple threads*. To hide the latency within a single thread, the request for the data (i.e. the prefetch request) must be moved back sufficiently far in advance of the use of the data in the execution stream. This effectively requires the ability to *predict* what data is needed ahead of time. In contrast, the multithreading approach splits read transactions by swapping out the currently executing thread when it suffers a miss, executing other concurrent threads for the duration of the miss to keep the processor busy, and eventually resuming the initial thread after the memory access completes.

In the context of software DSMs, the primary advantage of prefetching is that it does not consume additional parallel threads simply for the sake of hiding latency. This is appealing because achieving good speedups on a large number of threads can be quite challenging on a software DSM, and hence we do not want to assume that parallelism is abundant. In contrast, the advantages of multithreading include the fact that it can directly tolerate synchronization (as well as memory) latency, it does not rely on prediction (and hence can handle arbitrarily complex and unpredictable access patterns), and it does not require program modifications (unlike software-controlled prefetching). Both approaches involve run-time overheads, either to issue prefetches or to perform context switches. Given the strengths and weaknesses of both approaches, the best overall approach to dealing with latency in software DSMs has remained an open question.

1.3. Objectives and Overview

The goal of this paper is to characterize the benefits and costs of prefetching and multithreading in a software DSM environment, using a consistent hardware platform and set of applications. While studies do exist which consider each technique in isolation [3, 7, 24], the results cannot be directly compared since the architectural assumptions, DSM software, etc. are different. In addition to considering each technique in isolation, we also present the first results which *combine* prefetching and multithreading for the sake of hiding software DSM latency. Our study is based on a complete implementation of both prefetching and multithreading within a software DSM, running on real hardware (none of the results presented here are from simulations).

The remainder of the paper is organized as follows. We begin in Section 2 by describing our experimental environment, including the DSM software, the hardware, and the benchmark applications. Next, in Sections 3 and 4, we consider prefetching and multithreading in isolation, respectively. Section 5 evaluates the performance when the two techniques are *combined*. Finally, we discuss related work and present conclusions in Sections 6 and 7.

2. Experimental Framework

This section briefly describes the hardware and software used throughout our experiments.

2.1. DSM Software Layer: TreadMarks

All of our experiments are built on top of TreadMarks [12], which is a state-of-the-art software DSM implementation. As mentioned earlier, TreadMarks uses *lazy release consistency* (LRC) [13] and a *multiple-writer protocol* to minimize communication traffic. In this subsection, we briefly discuss some implementation details on TreadMarks which are relevant to later sections when we discuss how prefetching and multithreading are added (further details on TreadMarks can be found in Keleher *et al.* [12]).

TreadMarks uses a distributed timestamp and interval-based algorithm for maintaining LRC. Synchronization operations are explicitly labeled as either *acquires* or *releases*, and they define the boundaries of *intervals*, which processors designate by incrementing local timestamps. When synchronization occurs, the releasing processor piggybacks a vector timestamp (one element per processor) and *write notices* (to indicate which pages have been modified) along with the synchronization reply message. The acquiring processor then invalidates all pages for which a write notice is received. If one of these pages is subsequently accessed, the fault handler sends out messages to get an up-to-date copy of the page.

To avoid the ping-pong effects of false sharing, TreadMarks allows multiple processors to write to the same page simultaneously without interfering with each other. This is accomplished by later merging together runlength encoded records ("*diffs*") which are created by comparing the modified versions of the pages with clean, unmodified copies ("*twins*"). These *diffs* are applied to the shared pages according to the *happen-before-1* [1] partial order among the intervals (i.e. in increasing timestamp order) to ensure program correctness.

2.2. Hardware Platform

We performed our experiments on a collection of eight IBM RS/6000 workstations running AIX 4.1. Each workstation contains a 133 MHz PowerPC 604 processor, with split 16KB primary instruction and data caches, a 512KB unified secondary cache, and 96MB of physical memory. The machines were connected by a single FORE Systems ASX-200WG ATM LAN switch using 155 Mbps OC3 multimode fiber optic links. The TreadMarks processes communicate using a lightweight reliable communication protocol built on top of UDP. All timing measurements were done using the high-resolution timers provided by AIX 4.1.

2.3. Applications

We performed our experiments on the following set of applications: FFT, LU-NCONT, LU-CONT, OCEAN, RADIX, WATER-NSQ, and WATER-SP from the SPLASH-2 suite [26], and SOR from the TreadMarks distribution. FFT performs a 1D complex Fast Fourier Transform on 256K data points. LU-NCONT solves a blocked LU factorization of a 1024×1024 matrix with a block size of 128 where each block is allocated *non-contiguously*. LU-CONT solves the same problem as LU-NCONT except with a block size of 32 and *contiguously* allocated blocks. OCEAN simulates large-scale ocean movement based on eddy and boundary currents within a 258×258 grid. RADIX performs an integer radix sort with 2^{20} keys and maximum key value of 2^{21}. SOR performs a red-black successive over-relaxation on a 2000×2000 array over 50 iterations. WATER-NSQ simulates forces and potentials among 512 water molecules in liquid states across 9 time steps using an $O(n^2)$ algorithm. WATER-SP performs the same simulation as WATER-NSQ except with 4096 water molecules and an $O(n)$ algorithm. Further details on these applications can be found in studies by Woo *et al.* [26] and Liviu *et al.* [11].

3. Prefetching

We begin our study by focusing on prefetching alone. The idea behind prefetching is to use knowledge of future access patterns to bring remote data into the local memory before it is actually needed. In particular, we focus on *software-controlled* prefetching, where explicit prefetch calls are inserted into the code by either the programmer or the compiler. Since purely hardware-controlled prefetching probably does not make sense in a software DSM, the more realistic alternative to our approach is to have the DSM run-time layer issue prefetches automatically, perhaps based on access pattern histories [3]. While the advantage of this latter approach is that it does not require source code modifications, our experience has shown that by inserting prefetches explicitly, we can prefetch more intelligently and more aggressively [5].

In a multiprocessor environment, prefetches can be classified as being either *binding* or *non-binding*. With binding prefetching, the value seen by a subsequent read access is bound at the time when the *prefetch* operation completes. While binding prefetching is often easy to implement, it has the unfortunate property that the value may become stale if another processor modifies the same location during the interval between the prefetch and the read access. This places significant restrictions on where prefetches can safely be inserted. In contrast, with non-binding prefetching the data is brought close to the processor, but remains visible to the underlying coherence protocol such that the actual access is guaranteed to get the latest copy of the data. Hence the non-binding property gives the programmer or the compiler the flexibility to insert prefetches more aggressively without worrying about violating program correctness [20, 22]. Therefore we focus on *software-controlled non-binding prefetching* throughout this study.

3.1. Prefetching Implementation

Adding non-binding prefetching support to TreadMarks turns out to be non-trivial, due to the lazy and distributed nature of the consistency protocol. Unlike traditional directory-based protocols, where one can always get an up-to-date copy of memory by sending a message to the *home* node [16], determining the current set of modifications to a page at a given instant is difficult in TreadMarks since write notices are distributed across processors and are only known precisely at acquire time. The timestamping protocol was designed to provide correct information relative to the most recent synchronization point, but non-binding prefetches can be issued *prior* to synchronization.

We implemented non-binding prefetches in TreadMarks as follows. When a prefetch operation is executed, we first examine the set of write notices that have propagated to the local processor to determine which of them are more recent than the local copy of the page. For each of these write notices, we issue prefetch request messages to the corresponding nodes. Upon receiving a prefetch request, the servicing node replies with the necessary modifications, and write-protects the given page. If that page is subsequently modified, a new interval is created so that the coherence protocol can distinguish modifications before and after a prefetch reply. (The net result is that the intervals which are dictated by synchronization points may be further broken down into smaller intervals.) The prefetched data is stored in a separate heap (managed by the TreadMarks garbage collector), which can be thought of as a cache of remote *diff* replies. At the time when the page is accessed, these prefetched modifications will be applied to the page to bring it up to date. If the page has been modified by another processor since the prefetch, we simply request these remote *diffs* as normal, and apply them to the page *after* the prefetch modifications. Therefore the prefetches are truly non-binding, and never violate program correctness. (Further details on our prefetching implementation can be found in another publication [5].)

Since our prefetch operation only makes use of the existing write notices to fetch modifications, its effectiveness is limited by the lazy propagation of write notices in LRC. However, as we will see later in this section, this limitation does not have a large impact on application performance in practice.

When a prefetch is executed, we immediately check to see whether the page is already up-to-date in local memory, or whether remote requests for the updates are already in flight (either from earlier page faults or prefetches of the same page). If so, then the prefetch is dropped. Finally, it is important to note that prefetch requests are unreliable and may be dropped in the network.[3] Therefore if a prefetch fails to return before a page fault occurs on the real access, we do not wait for the prefetch (since it may never return), but instead issue a normal remote memory request at that time.

3.2. Inserting Prefetches

We inserted explicit prefetch procedure calls into the source code of the applications as follows. With the exception of WATER-SP, all of the other applications use arrays as their primary data structures (WATER-SP uses linked lists). Therefore we apply Mowry's prefetching algorithm [20] to these applications to isolate dynamic miss instances through loop-splitting techniques (e.g., strip mining) and to schedule prefetches far enough ahead using software pipelining. Prefetching for software DSM is quite similar to prefetching page faults to hide the latency of out-of-core I/O [21]. For two of the seven array-based applications (FFT and LU-NCONT), our implementation of prefetching in the SUIF compiler [25] achieved performance comparable to the best that we could do by hand; in the other five cases (LU-CONT, OCEAN, RADIX, SOR and WATER-NSQ), we achieved better performance through hand-tuning. Hence to show the full potential of prefetching, we use hand-inserted prefetching in the latter five cases and compiler-inserted prefetching for FFT and LU-NCONT.

For WATER-NSQ in particular, the non-binding property of our prefetches is important. WATER-NSQ is a multiple-producer, multiple-consumer application where the major misses occur when updating shared locations protected by locks. With non-binding prefetching, we can insert prefetches *before* the locks, thereby

[3] One of the problems with making prefetches reliable is that during high network congestion, we do not want to continuously retry sending a prefetch, since this would make the congestion even worse. With our scheme, we will retry only once (upon the actual access) if the prefetch fails.

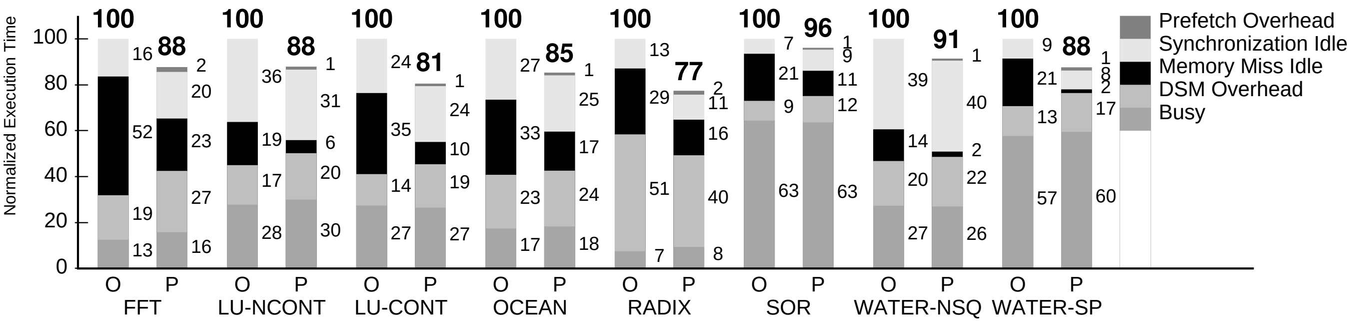

Figure 2. Performance impact of prefetching (O = original, P = with prefetching).

giving us more time to hide the latency.

We inserted prefetches into WATER-SP (a pointer-based program) by hand using a variation of the *history prefetching* scheme proposed by Luk and Mowry [19]. Since the recursive data structures do not change once they are created, we create a new local array and use it to record pointers to the elements in the traversal order. Therefore to issue prefetches early enough, we simply dereference the pointers in this local array, thereby circumventing the pointer-chasing problem.

3.3. Prefetching Results

Figure 2 shows the impact of prefetching on execution time for all of the applications. For each application, the topmost portion of the normalized execution time for the prefetching case (labeled "*Prefetch Overhead*") represents the overhead of issuing the prefetches. The remaining categories are the same as in Figure 1, except that for the prefetching case, the "*Busy*" time includes prefetch overheads associated with loop transformation and unnecessary prefetches.[4] "*DSM overhead*" includes the amount of time spent servicing prefetch requests and replies.

As we see in Figure 2, prefetching improves the execution time of all of these applications, with speedups ranging from 4% to 29%. This improvement is due to significant reductions in the memory miss stall times, ranging from 45% to 92%. This benefit is large enough that it more than offsets the runtime overheads of prefetching. Despite the fact that each prefetch which generates a remote message (i.e. those that are not immediately dropped because they are unnecessary) requires roughly 140 μsec of software overhead, the "*prefetch overhead*" category remains quite low (under 3%) in all cases. The "*DSM overhead*" increases somewhat due to two effects: (i) prefetch requests are more expensive to service than normal memory requests, since they involve creating new intervals when the requested page is dirty; and (ii) prefetches which fail to fully hide the latency result in a retry request upon the real access. The "*busy*" times generally increase due to the extra prefetch computation embedded in the application source codes and due to unnecessary prefetches. The slight decrease in busy time in some cases is largely due to small measurement variations, since we are running on real hardware. Finally, we see that while prefetching typically does not have much impact on synchronization time, RADIX does enjoy a 21% reduction due to improved load balancing.[5]

While the performance improvements offered by prefetching are substantial, a natural question is why did prefetching fail to hide *all* of the latency? To develop a deeper understanding of the limitations of prefetching in a software DSM, we will focus on two issues: (i) the success of the prefetch insertion strategy in selecting the appropriate references to prefetch and scheduling them early enough, and (ii) the effects of network traffic.

3.3.1. Prefetching Effectiveness

To evaluate the effectiveness of our prefetches, two concepts are useful: the number of *unnecessary prefetches* and the *coverage factor*. Unnecessary prefetches are prefetch operations which find their data locally. The coverage factor is the fraction of original remote misses that are prefetched. An ideal prefetching scheme will have a 100% coverage factor and no unnecessary prefetches. Table 1 shows the percentage of unnecessary prefetches and the coverage factors.

If we focus on unnecessary prefetches, we see that five of the eight applications have roughly 50% or more unnecessary prefetches. The numbers are particularly

[4] The overhead associated with an unnecessary prefetch involves an address lookup, checking the "valid" flag for the given page locally, and a conditional branch.

[5] Part of this effect was reduced garbage collection time. Prefetching reduces garbage collection times by allowing the garbage collector to validate dirty pages more quickly.

Table 1. Prefetching statistics (O = original, P = with prefetching).

Benchmark	Unnecessary Prefetches	Coverage Factor	Total Traffic (KBytes)		Total Misses		Average Miss Latency (μsec)	
			O	P	O	P	O	P
FFT	98.38%	99.89%	63112	64395	13013	464	3600	43800
LU-NCONT	47.93%	88.80%	145024	140292	33002	4052	2400	5800
LU-CONT	11.48%	94.54%	35800	36509	6008	550	3900	11700
OCEAN	35.44%	74.41%	210688	218658	71464	25399	2000	2900
RADIX	11.39%	94.27%	141477	146402	16437	3359	3800	10000
SOR	12.48%	99.96%	14683	15383	4917	166	3200	50698
WATER-NSQ	77.91%	91.36%	23671	24140	9483	1230	1600	2100
WATER-SP	79.32%	97.61%	64641	65932	21724	1387	1721	2000

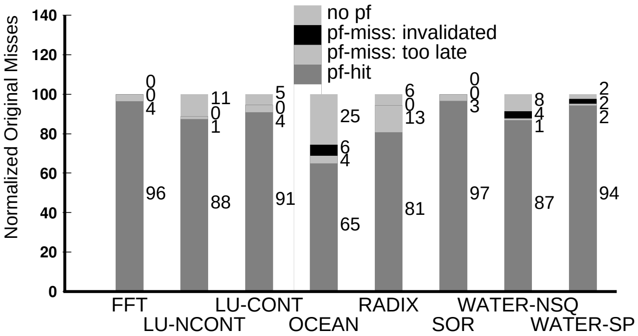

Figure 3. Breakdown of the original remote misses.

high in the compiler-inserted prefetching cases (FFT and LU-NCONT) since the compiler cannot distinguish private from shared memory locations, and hence wastes prefetches on private data. However, since the prefetching overheads in Figure 2 are generally quite small, wasted overhead is clearly not the real problem.

The real question is why the *upside* of prefetching is not larger. Based on the large coverage factors in Table 1, we would expect to see larger reductions in the memory stall times. For example, 99.9% of the original remote misses in FFT were prefetched, but nearly half of the original memory stall time still remains. To help answer this question, Figure 3 shows a more detailed breakdown of what happened to the original remote misses. The topmost section ("*no pf*") is the fraction that was not prefetched. The remaining three cases constitute the coverage factor. The ideal case is "*pf-hit*", where the prefetches fully hide the latency—notice that this is the largest case for all applications.

The "*pf-miss: invalidated*" case is where prefetched data is brought into the local memory but is invalidated before it can be used. This situation arises because we use *non-binding* prefetches, and therefore sometimes the data must be invalidated to preserve correctness. This case is generally small, and is most noticeable in OCEAN and WATER-NSQ where prefetches are moved back across barriers and locks, respectively. Although this is an interesting effect, the fact that the invalidations allow prefetches to be non-binding results in a performance benefit that far outweighs the cost of these occasionally unsuccessful prefetches.

Finally, the "*pf-miss: too late*" category are cases where the data has been prefetched but does not return in time to satisfy the reference. With the exceptions of OCEAN and WATER-NSQ, this case accounts for most of the unsuccessful prefetches. There are two reasons for this problem: either the prefetches were not issued early enough to hide the latency, or else the prefetch request messages were dropped in the network. Both of these cases are affected by the network traffic, which we now consider in greater detail.

3.3.2. Network Traffic

The pair of columns in the center of Table 1 compare the total number of bytes sent through the network in the original and prefetching cases. Although we generally see an increase in the prefetching case,[6] this increase is typically quite small, and hence does not suggest a problem. The rightmost pairs of columns in the table show the impact of prefetching on the number of misses and their average latency. In all cases, the number of remote misses has been reduced. However, in several cases (FFT, LU-CONT, RADIX, and SOR) the average miss latency has increased *enormously*. For example, prefetching has reduced the number of misses in FFT by a factor of 28, but has increased the average miss latency by a factor of 12. We observe that this problem is due to the burstiness of traffic in the network, which is causing extreme queueing delays and/or dropped messages. In particular, we see hot-spotting effects during program initialization (when all proces-

[6]Except in LU-NCONT, where the separate prefetch heap provides additional storage area for the diffs and therefore reduces the need for garbage collection, thus reducing memory traffic.

sors are trying to communicate with a single processor) which are particularly acute.

Overall, we observe that software-controlled non-binding prefetching can yield substantial performance improvements (4% to 29% speedups) by reducing memory stall times. Despite high coverage factors, one of the limitations which prevents prefetching from doing better is network contention delays caused by attempting to compress the original message traffic into a small period of time, and also due to hot-spotting. Another limitation of prefetching is that it does not directly address synchronization latency, which can be quite significant in software DSM (as we saw in Figure 2).

4. Multithreading

To tolerate the latency of synchronization as well as remote memory accesses, we now consider multithreading. The idea behind multithreading is to switch from one parallel thread to another upon a long-latency operation, thereby keeping the processor busy with useful work until the remote operation completes. The performance of multithreading depends on several factors, including whether there is enough parallelism in the application such that a ready-to-run thread is always available, whether a significant amount of time is wasted during thread switches, and whether the locality effects of sharing the same local portion of the memory hierarchy are positive or negative.

4.1. Multithreading Implementation

We implemented multithreading on top of TreadMarks using the Pthreads [10] user-level thread library. The benefit of user-level threads is that since the threads share the same memory image within a processor, there is less state to save on a context switch, and there is less overhead managing the local portion of shared memory. For example, if one thread brings a page into the local portion of shared memory, another thread can also use it directly. Hence if there is significant locality within the clusters of threads on each processor, we can see a prefetching benefit relative to running them on distinct processors.

In our implementation, a thread switch occurs whenever the current thread encounters a long latency event—i.e. a remote memory miss or a remote synchronization operation. We do not necessarily restart threads immediately once the event they were waiting for completes—instead, we will mark the thread as "ready to run," and will potentially restart it whenever the current thread is swapped out.

The main overhead of multithreading within software DSM is a larger number of asynchronous message arrivals. Without multithreading, a process can often spin on a particular message queue, waiting for a reply to its outstanding request. With multithreading, however, spinning no longer occurs since a thread typically switches control to another thread upon long-latency events. Hence there is non-trivial kernel overhead due to signaling as messages arrive asynchronously.

To avoid unnecessary communication, we combine outstanding requests whenever possible. For remote memory accesses, we simply stall a subsequent access to the same page until the reply returns and the page is validated. For locks, we do something similar, except that we still maintain mutual exclusion on the lock among local threads once it returns (of course). By keeping track of which local threads are queued waiting for a given lock, we can pass the lock very quickly between threads on the same processor. Combining requests for barriers is somewhat different—we gather the local arrivals first such that only the last local thread to arrive generates a remote arrival message. (Further details on our multithreading implementation can be found in another publication [18].)

4.2. Modifications to the Applications

Most applications do not require any modifications to run correctly in a multithreading fashion, other than replicating "private" data on the heap whenever appropriate such that each thread has its own copy. However, in cases where a significant amount of redundant computation is performed initializing these private copies of data structures, a useful performance optimization is to keep a single shared copy of the data structure per processor, thus avoiding wasted computation and synchronization. We applied this latter optimization to FFT and WATER-NSQ, resulting in large performance improvements in both cases.

4.3. Multithreading Results

Figure 4 shows the performance impact of multithreading with two, four, and eight threads per processor relative to the original execution time. The "*Multithreading Overhead*" component of the normalized execution time represents the overhead for switching between threads—all other components are the same as in Figure 1. In addition, the "*DSM overhead*" includes any time spent servicing asynchronous message arrivals and combining remote requests.

As we see in Figure 4, multithreading improves the performance of six of the eight applications by 6% or more, with two applications speeding up by over 50%. The optimal number of threads varies across the board. To provide further insight, Table 2 shows several statistics on the behavior with multithreading. Using the

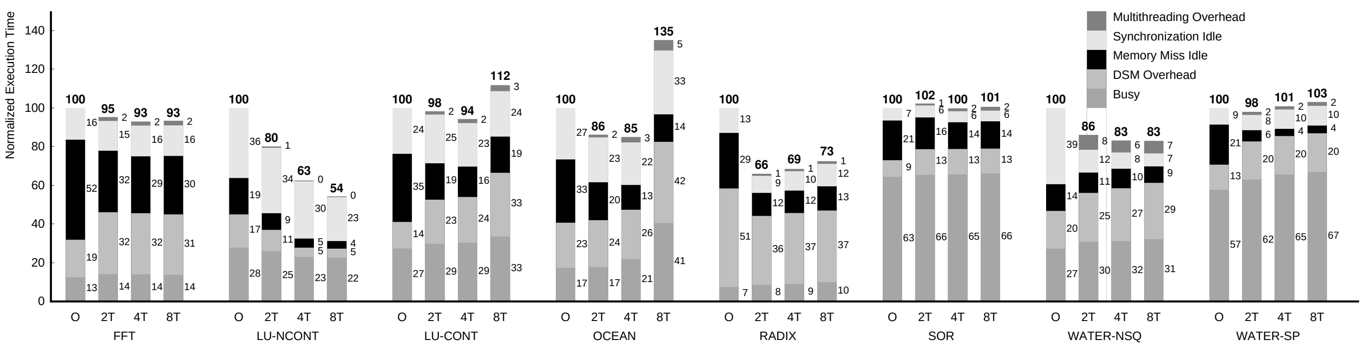

Figure 4. Performance impact of multithreading (O = original, nT = multithreading with n threads)

Table 2. Multithreading statistics (O = original, nT = multithreading with n threads).

Benchmark		Avg. Stall Time (μsec)	Avg. Run Length (μsec)	All Messages Total (Msgs)	All Messages Volume (Kbytes)	Remote Misses Total (Msgs)	Remote Misses Avg. Stall (μsec)	Remote Locks Total (Msgs)	Remote Locks Avg. Stall (μsec)	Remote Barriers Total (Msgs)	Remote Barriers Avg. Stall (μsec)
FFT	O	3554	856	31520	63112	13013	3554	0	0	70	200000
	2T	2189	967	31520	63279	13013	2189	0	0	70	185714
	4T	2020	959	31520	63279	13013	2020	0	0	70	200000
	8T	2072	943	31519	63278	13013	2072	0	0	69	202898
LU-NCONT	O	2350	3465	156067	145024	33002	2350	0	0	250	596000
	2T	1790	3443	66508	99605	19829	1790	0	0	198	666666
	4T	1823	5271	25902	44438	10623	1823	0	0	190	642105
	8T	1534	5436	25455	46802	10403	1534	0	0	177	531073
LU-CONT	O	3902	2788	18777	35800	6008	3902	0	0	250	596000
	2T	2080	2614	18912	36169	6004	2080	0	0	198	666666
	4T	1738	2432	18980	36306	5990	1738	0	0	190	642105
	8T	1380	2338	29540	57598	9030	1380	0	0	177	531073
OCEAN	O	2094	936	167918	210688	71464	1970	1645	7506	7209	14010
	2T	1424	1051	149178	160241	62281	1356	1656	4003	7208	12902
	4T	940	1161	151816	125359	64212	858	1657	4155	7208	12069
	8T	628	1289	250718	198124	112278	545	1662	6287	7208	18174
RADIX	O	3776	979	262400	141477	16437	3788	93	1680	141	191489
	2T	1480	1049	160008	97029	17490	1483	105	1077	133	135338
	4T	1422	1097	160386	97671	17650	1430	126	369	134	156716
	8T	1496	1151	161142	98963	17970	1503	167	266	134	186567
SOR	O	3184	8614	11258	14683	4917	3184	0	0	800	5000
	2T	2550	8729	11258	14723	4917	2550	0	0	800	5000
	4T	2144	8758	11258	14723	4917	2144	0	0	800	5000
	8T	2185	8821	11258	14723	4917	2185	0	0	800	5000
WATER-NSQ	O	1507	847	91997	23671	9483	1610	25808	1470	440	11363
	2T	509	1012	91576	24146	9425	1244	25806	241	440	13636
	4T	393	1727	91476	24265	9393	1195	25808	102	440	13636
	8T	292	3067	91578	24429	9419	997	25810	36	440	13636
WATER-SP	O	1873	4704	47948	66191	21724	1721	386	10432	440	25000
	2T	519	5116	47363	66670	21655	487	253	3284	440	79545
	4T	338	5222	47474	66881	21677	306	249	3124	440	36363
	8T	354	4733	47363	66702	21659	325	252	2849	440	38636

first two columns in Table 2, one can roughly estimate the number of additional threads needed to hide the latency (under ideal circumstances) by dividing the average stall time by the sum of the average run length plus the average context switch time (which is roughly 110 μsec). For example, this ratio would suggest that two threads should suffice for WATER-SP—as we see in Figure 4, WATER-SP does achieve its best performance with two threads. In most cases, however, the performance is dominated by other effects.

The impact of multithreading on locality—which translates into variations in network traffic (see Table 2)—has both positive and negative effects on performance. LU-NCONT and RADIX enjoy improved locality with multiple threads, while locality degrades in LU-CONT and OCEAN beyond four threads. Since each processor's local memory is large enough to hold the entire data set of each application, the major effect is how multithreading affects communication, particularly due to *false sharing*. On the one hand, multithreading can result in better task assignments which improve spatial locality (LU-NCONT); on the other hand, reducing the block size too much to accommodate additional threads can induce false sharing (OCEAN).

Another limiting factor is bursty misses along with hot-spotting in the network. This was particularly problematic in FFT and SOR, where hot-spotting occurs as the processors read their initial data sets from the master processor. Since performance is limited by the throughput of master processor's network link in these cases, multithreading shows less improvement than one might otherwise expect.

Finally, the overhead of multithreading can offset a significant fraction of the latency tolerance benefit in some cases (e.g., WATER-SP). Note that the major effect is not context switching time, but rather the overhead of handling asynchronous message arrivals (which appears as "*DSM Overhead*" in Figure 4).

5. Combining Prefetching and Multithreading

Having considered prefetching and multithreading in isolation, we now focus on *combining* these techniques. Whether the combination of both techniques offers better performance than either technique alone on a software DSM has remained an open question. On the one hand, each technique might compensate for the other technique's weaknesses, thereby hiding more latency; on the other hand, the techniques may interfere with each other, thus degrading performance.

How should prefetching and multithreading be combined? To hide synchronization latency, one might expect multithreading to be the right answer, since prefetching does not directly address this problem. To hide memory latency, however, the right approach is less clear, since both prefetching and multithreading can potentially hide this same latency.

One approach is to apply multithreading to synchronization latency, and to apply *both* prefetching and multithreading to memory latency. We experimented with this approach, but found that despite all of our efforts to improve its performance (including the optimizations described later in Section 5.1), it never achieved better performance than either prefetching or multithreading alone [5, 18]. The problem with this approach is that switching between threads tends to result in bursty miss patterns, which in turn slow down requests in the network, including prefetches. When prefetches fail to return in time, a retry occurs (since prefetches are unreliable) followed by a thread switch, thereby further exacerbating the problem. As a result, we tend to pay the full overheads of *both* prefetching and multithreading (in fact, the overhead tends to go up due to more asynchronous message arrivals), without appreciably improving our ability to hide latency (in part because of increased queueing delays).

Therefore in the remainder of this section, we consider a different approach to combining prefetching and multithreading. We apply multithreading only to synchronization latency, and we use prefetching to hide memory latency.

5.1. Optimizing Prefetching for Multithreading

We discovered that naively applying our original prefetching scheme to multithreaded code resulted in disappointing performance for the following reason. Since there is often significant overlap among the working sets of threads running on the same processor, the first thread which touches remote data often effectively "prefetches" it for other threads. Hence we would like to avoid having these subsequent threads issue unnecessary prefetches, but there are two complications: all threads execute the same static code, and we do not know *a priori* which thread will arrive at the data first (since threads are scheduled dynamically). To address this problem, we identify cases where threads on the same processor would be redundantly prefetching the same data, and we protect these prefetches with a conditional test of a dynamic flag which is explicitly reset by the first processor to arrive at the data.

A second optimization was useful in the case of RADIX. To help reduce the load on the network, we throttled back the number of prefetches (in this case by eliminating every-other dynamic prefetch). Although this resulted in a lower prefetching coverage factor, this was more than offset by reductions in network queueing delays in this particular case.

5.2. Performance of the Combined Approach

Figure 5 shows the impact of combining prefetching and multithreading on performance. In three of the eight applications (FFT, OCEAN and WATER-NSQ), we see that the best performance is achieve through the combination of both techniques, with speedups ranging from 4% to 26% over either technique alone. In two cases (LU-NCONT and RADIX) the best performance occurs with multithreading alone, and in three cases (LU-CONT, SOR and WATER-SP) prefetching alone performs the best.

Why does the combined approach fail to outperform the individual techniques in these latter five cases? In LU-NCONT, the combined case actually does achieve

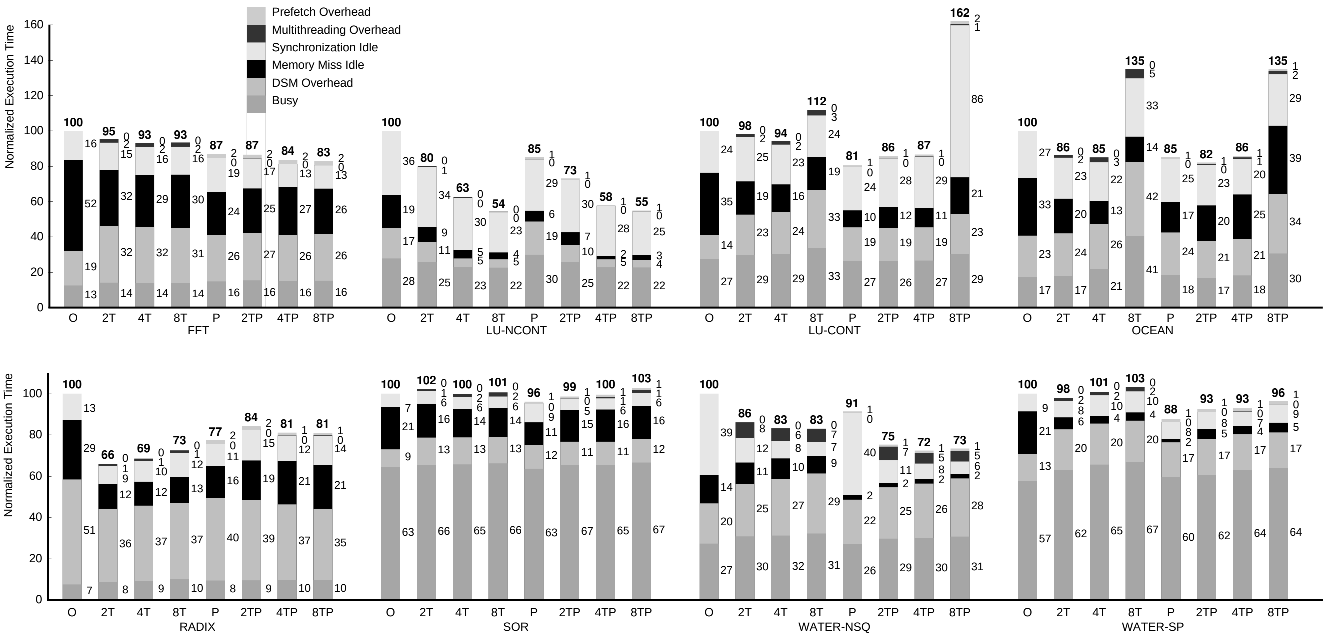

Figure 5. Performance impact of combining prefetching and multithreading (O = original, P = with prefetching, nT = n threads without prefetching, nTP = n threads with prefetching)

the best performance with fewer than eight threads per processor, and is comparable to the best case even with eight threads.

In RADIX, the primary problem is that the loop structure makes it difficult to schedule prefetches early enough to hide the large network latencies—notice in Figure 3 that RADIX has the largest fraction of late prefetches. In addition, RADIX suffers from network contention delays due to its very high rate of communication. These two effects interact negatively: increased network contention slows down prefetches, causing them to arrive even later, and as more prefetches become late, they result in more retry messages, thus resulting in more network contention. Hence multithreading alone is the clear winner for RADIX, since it can address memory latency without requiring the ability to predict addresses far in advance (thus avoiding the late prefetch problem).

In LU-CONT, SOR, and WATER-SP, prefetching is more effective at hiding memory latency than multithreading, so clearly we would at least want to use prefetching. What may be surprising, however, is that we are better off doing nothing for synchronization latency rather than attempting to tolerate it through multithreading. The reason for this is twofold. First, the bulk of the synchronization latency in these cases is due to barriers, and multithreading improves barrier stall times only indirectly by improving load balance. Second, there is a significant fixed cost involved in supporting multithreading, since all message arrivals must then be handled asynchronously. As we see in Figure 5, this increased overhead (which mostly appears as "*DSM Overhead*") more than offsets any gain in reduced synchronization stall times.

The largest performance gain from the combined scheme is in WATER-NSQ. In this case, prefetching eliminates over 80% of the memory stall time (as opposed to less than 50% with multithreading), and multithreading eliminates roughly 80% of the synchronization stall time (which is primarily due to locks). Hence WATER-NSQ is a good illustration of how prefetching and multithreading can be combined in a complementary fashion.

5.3. Summary

What is the best overall strategy for tolerating latency in a software DSM? If it is difficult to predict memory addresses early enough to schedule prefetches effectively (e.g., RADIX), then multithreading alone may be the best solution, although one must be careful not to reduce block sizes to the point where they induce false sharing. Once you pay the overhead of supporting multithreading to hide memory latency, there is no reason not to also apply it to synchronization latency.

If addresses are predictable enough that prefetches can be scheduled sufficiently far in advance, we observe that prefetching is generally just as good (if not better) than multithreading at tolerating memory latency. We found no case where it was best to apply both prefetching and multithreading to memory latency—instead, one should choose either one scheme or the other.

In cases where prefetching is used to tolerate memory latency, it is most likely that multithreading will complement prefetching (by hiding synchronization latency) if lock stalls account for a significant fraction of execution time. On the other hand, if synchronization stall times are small or are dominated by barrier stalls, it is less clear that the additional overhead of supporting multithreading will be worthwhile.

6. Related Work

Both prefetching and multithreading have been studied previously in the context of tightly-coupled multiprocessors [2, 8, 9, 14, 15, 20]. Prefetching for software DSMs has been studied by Dwarkadas *et al.* [7] and by Bianchini *et al.* [3]. The former study focused on compilation techniques to automatically insert prefetches into numeric applications. The latter study examined *binding* prefetches which were launched at synchronization points based on access pattern histories. The results in this latter study demonstrated that binding prefetching results in bursty traffic, it increases synchronization time, and is not appropriate for locks that protect small critical sections. In contrast, our non-binding prefetches can be moved back ahead of locks, and therefore do not suffer from these same problems.

Thitikamol and Keleher [24] studied the impact of multithreading on software DSMs. They found that multithreading could improve application performance with a small number of threads. They also observed that the speedup is often limited by contention for local resources, and argued that reduction operations should be explicitly identified in the source code to achieve better performance.

In contrast with these earlier studies, our work is the first to evaluate *both* techniques using a consistent hardware platform and set of applications, thereby allowing direct comparisons. In addition, this is the first study to consider *combining* prefetching and multithreading in a software DSM.

An interesting comparison is between our results and the results of the earlier study by Gupta *et al.* [8] on combining prefetching and multithreading in a tightly-coupled multiprocessor. Similar to their study, we also conclude that combining prefetching and multithreading produces mixed results. In contrast with that study, we have found that prefetching and multithreading work best together in a software DSM when multithreading focuses primarily on synchronization latency, and allows prefetching to handle remote memory latency.

7. Conclusions

This paper has focused on how prefetching and multithreading, both individually and in combination, can address the communication latency bottleneck in software DSM systems. We performed our experiments on real hardware using a full implementation of both techniques.

We found that that software-controlled non-binding prefetching offers significant performance improvements by hiding roughly 50% or more of the memory stall times for most of the applications we consider. The reason why prefetching does not achieve even better performance is that network contention is greatly increasing the latency of the references which are not successfully prefetched.

Multithreading addresses not only remote memory latency, but also synchronization latency. The overall speedups from multithreading alone were greater than 50% in two cases, which is larger than the best improvement that we saw from prefetching alone. To get the best performance from multithreading, the applications should be modified to take local sharing patterns into account.

By combining both prefetching and multithreading such that multithreading hides synchronization latency and prefetching hides memory latency, we found that three of the eight applications can achieve better performance than when we use either technique individually. We do observe, however, that combining prefetching and multithreading such that both techniques attempt to hide memory latency is generally not a good idea, and hurts performance through redundant overhead in most cases. The best overall approach to hiding latency depends on factors such as the predictability of memory access patterns and the extent to which lock stalls dominate synchronization time.

8. Acknowledgments

This work is supported by the Information Technology Research Centre of Ontario and by grants from the Natural Sciences and Engineering Research Council of Canada. Todd C. Mowry is partially supported by a Faculty Development Award from IBM.

References

[1] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. Technical Report

CS1051, University of Wisconsin, Madison, September 1991.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. D. Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[5] C. Q. C. Chan. Tolerating Latency in Software Distributed Shared Memory Systems Through Non-Binding Prefetching. Masters Thesis, Department of Computer Science, University of Toronto, *forthcoming*.

[6] A. L. Cox, S. Dwarkadas, and P. Keleher. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 106–117, April 1994.

[7] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[8] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, May 1991.

[9] R. H. Halstead, Jr. and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[10] IEEE. Threads Extension for Portable Operating Systems (Draft 7), February 1992.

[11] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[12] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter Usenix Conference*, January 1994.

[13] P. Keleher, A. L. Cox, and W. Zwaepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[14] J. S. Kowalik, editor. *Parallel MIMD Computation : The HEP Supercomputer and Its Applications.* MIT Press, 1985.

[15] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[16] J. Laudon and D. Lenoski. The SGI Origin: A cc-NUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[17] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, 7(4):321–359, November 1989.

[18] A. K. W. Lo. Tolerating Latency in Software Distributed Shared Memory Systems Through Multithreading. Masters Thesis, Department of Computer Science, University of Toronto, *forthcoming*.

[19] C. K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[20] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Stanford University, March 1994.

[21] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[22] T. C. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared memory Multiprocessors. *Journal of Parallel and Distributed Computing*, June 1991.

[23] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[24] K. Thitikamol and P. Keleher. Multi-threading and Remote Latency in Software DSMs. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997.

[25] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. *Languages and Compilers for Parallel Computing*, pages 137–151. Springer-Verlag, Berlin, Germany, 1992.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.