Per-Node Multi-Threading and Remote Latency

Kritchalach Thitikamol and Pete Keleher
University of Maryland
(kritchal | keleher)@cs.umd.edu

Abstract

This paper evaluates the use of per-node multi-threading to hide remote memory and synchronization latencies in software DSMs. As with hardware systems, multi-threading in software systems can be used to reduce the costs of remote requests by running other threads when the current thread blocks.

We added multi-threading to the CVM software DSM and evaluated its impact on the performance of a suite of common shared memory programs. Multi-threading resulted in speed improvements of at least 20% in two of the applications, and better than 15% for several other applications. However, we also found that good performance can not always be achieved transparently for non-trivial applications. Also, the characteristics of the underlying DSM protocol can have a large effect on multi-threading's utility.

1. Introduction

This paper presents an empirical evaluation of the use of per-node multi-threading to hide remote latencies the CVM [1] software distributed shared memory (DSM) system. DSMs are software systems that emulate shared memory semantics in software over hardware that provides support only for message-passing. Multi-threading for latency-hiding is a well-known technique for hiding cache miss latencies in the hardware environment [2, 3]. However, the software environment presents special challenges.

The paradigm usually assumed in DSM-related literature is that of a distributed system containing a single thread on each processor. This arrangement is simple, and yet allows reasonably high processor efficiency. However, DSMs often have high remote communication latencies, causing the performance of such systems to be largely dependent on the frequency with which remote lock acquires or data requests occur. Although the portion of this latency contributed by local software overhead is often significant, the majority results from time on the wire and processing at the remote location.

To see that this is true, consider Figure 1. Request latency can be broken into local send overhead, L_s, wire time for the request, W_1, remote receive overhead, R_r, remote processing of the request, R_p, remote send overhead, R_s, wire time for the reply, W_2, and local receive overhead, L_r. In our environment, wire time is insignificant compared to send and receive OS overheads L_s, R_r, R_s, and L_r. Assuming that local and remote OS overheads are the same, the local processing

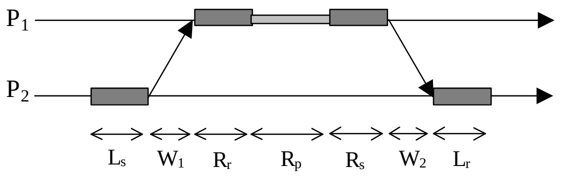

Figure 1: Local vs. Remote Latency

time, $L_s + L_r$, is less than half the total latency of the request. The rest of this latency can be used to run a second thread, provided sufficient parallelism is available.

The primary performance advantage of per-node multi-threading (MT) is that multiple threads can be used to ensure that work is available when the currently active thread stalls on a remote request. If the level of multi-threading is high enough, multiple threads can often hide all remote request latency other than local operating system overhead. Second, the separation of the system's virtual machine from the physical machine may allow a better mapping of the computation on to the thread model. Specifying the number of threads in an application independently of any particular architecture has several advantages, including architecture independence, clarity of expression, implicit load balancing, and ease of code generation for parallelizing compilers [4]. MT also has several disadvantages, mostly relating to the frictional cost of dealing with additional threads.

This paper has three main contributions. First, we measure the effect of MT on a state-of-the-art DSM protocol and identify factors that limit MT's performance improvement. Second, we assess how transparently MT can be added to applications, both in terms of correctness and in terms of performance. Finally, we test MT with two other protocols and identify protocol-specific factors that affect MT's performance advantages.

The rest of this paper proceeds as follows. Section 2 describes the programming model assumed by CVM, and the changes made to support multi-threading. Section 3 describes the implementation of multi-threading in CVM and the implications of various design choices. Section 4 describes our performance, and Section 5 concludes.

2. Programming Model

The majority of DSM-related literature assumes a location-transparent programming model in which the number of threads and processors is specified as part of the input. Application behavior other than performance is assumed to be independent of the number of system threads. Synchronization is usually accomplished through system calls to the DSM.

While not specified in the programming model, most of these systems locate a single thread on each physical processor in the system. The advantage of this scheme is its simple cost modal. More than a single thread per node would introduce frictional costs resulting from switching between the local threads.

Additionally, a single thread per node simplifies handling the scope of “global” application variables. CVM (and most other DSMs) make such variables non-shared. This is acceptable with a single thread per node because sharing relationships are uniform, i.e. each thread has private stack data and global variables. However, user-level threads on a single node all share the same view of global variables. This is problematic because it introduces an additional level in the thread hierarchy: threads on a single node share global variables, while threads on different nodes have distinct copies. Unfortunately, it is often quite difficult (and non-portable) to ensure that application globals, CVM globals, and other library globals are assigned to distinct pages. Our current solution is to disallow modifications to global data after initialization is complete. Since global data is consistent across all nodes until startup has finished, we thereby ensure that global data will be uniform across the views of all threads.

Multi-threaded nodes do add an additional level to the hierarchy of memory access times, i.e. threads that are colocated on a single node share an affinity that is not present between threads located on different nodes. To understand the problem, consider an example in which a simple matrix application allocates the computation in contiguous chunks of rows to each thread. With only a single thread per processor, the distribution automatically benefits from any spatial locality in the computation, as all rows on a single node are contiguous. In the multi-threaded case, care must be taken to allocate consecutive chunks of the matrix to threads on the same node. Otherwise, locality exploited by the single-threaded system is potentially not present in the multi-threaded case.

The division into multiple threads can be problematic even if contiguous matrix chunks *are* allocated to all threads on the same node. For example, consider the case where each thread moves linearly through its portion of the matrix, and there are two threads per node. If data is shared on the boundary between each pair of threads, the data on the boundary between two local threads will be accessed at the beginning of an iteration by the second thread, and at the end of the iteration by the first. Between the time of the second thread’s access and that by the first, the data may have been displaced from local memory because of consistency actions or lack of capacity. The multi-threaded system will then suffer more access misses than the single-threaded system, even though the same data is allocated to each node in both cases. This case will not affect correct even though performance might be compromised.

In both the single- and multi-threaded cases, threads synchronize through global locks and barriers. No process is allowed to proceed past a global barrier before all processes arrive. Global locks can be held by only a single thread at a time.

3. Implementation

3.1 CVM

The DSM target used in this work is CVM, a software DSM that supports multiple protocols and consistency models. Like commercially available systems such as [5], CVM is written entirely as a user-level library and runs on most UNIX-like systems. Unlike TreadMarks [5], CVM was created specifically as a platform for protocol experimentation.

The system is written in C++, and opaque interfaces are strictly enforced between different functional units of the system whenever possible. The base system provides a set of classes that implement a generic protocol, user-level threads, and network communication. The latter functionality consists of efficient, end-to-end protocols built on top of UDP.

New shared memory protocols are created by deriving classes from the base `Page` and `Protocol` classes. Only those methods that differ from the base class's methods need to be defined in the derived class. The core DSM routines call protocol hooks before and after page faults, synchronization, and I/O events. Since many of the methods are inlined, the resulting system is able to perform within a few percent of a severely optimized system, TreadMarks, running a similar protocol. However, CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration. We therefore expect the performance of the fully functional system to improve over the existing base.

Memory Consistency - CVM's primary protocol implements a multiple-writer version of lazy release consistency, which is a derivation of *release consistency*[6]. In release consistency, a processor delays making modifications to shared data visible to other processors until special *acquire* or *release* synchronization accesses occur. The propagation of modifications can thus be postponed until the next synchronization operation takes effect. Programs produce the same results for the two memory models, provided that all synchronization operations use system-supplied primitives, and that all conflicting shared accesses are ordered by synchronization or program order. In practice, most shared-memory programs require little or no modifications to meet these requirements.

Lazy release consistency (LRC) [7] allows the propagation of modifications to be further postponed until the time of the next subsequent acquire of a released synchronization variable. At this time, the acquiring processor determines which modifications it needs to see according to the definition of LRC. To do so, the execution of each process is divided into *intervals*, each denoted by an *interval index*. Potentially each synchronization operation causes a new interval to begin and the interval index to be incremented. Intervals of different processes are partially ordered by assigning a *vector timestamp* to intervals for each processor. At an acquire, processor *p* sends its current vector timestamp to the previous releaser of the same synchronization variable, *q*. Processor *q* then piggybacks on the release-acquire message to *p write notices* for all intervals named in *q*'s current vector timestamp but not in the vector timestamp it received from *p*.

False sharing - False sharing occurs when two or more processors access different variables within a page, with at least one of the accesses being a write. False sharing is problematic for software DSMs because of the large page-size coherence units. CVM's *multiple-writer* protocol reduces the effects of false sharing by allowing two or more processors to simultaneously modify local copies of the same shared page without prior negotiation.

These concurrent modifications are merged using *diffs* to summarize the updates. A diff is created by performing a page-length comparison between the current contents of the page and a *twin* of the page that was created at the first write access. If each concurrent writer summarizes its modifications as a diff, the system can create a copy that reflects all modifications by applying the concurrent diffs to the same copy. Concurrent diffs only overlap if the same location is written by multiple processors without intervening synchronization, which is probably a data race.

OS interface - CVM uses the UNIX `mprotect` system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a `SIGSEGV` signal (segmentation violation). The `SIGSEGV` signal handler examines local information to determine the page's state. If the local copy is read-only, the handler allocates a page from the pool of free pages and performs a `bcopy` to create a *twin*. Finally, the handler upgrades the access rights to the original page and returns. If the local page is invalid, the handler requests a copy from the page's owner. If write notices are present for the page, the faulting processor requests the corresponding diffs in parallel. After all necessary diffs have been received they are applied to the page in increasing timestamp order.

Multi-threaded CVM - We extended the original CVM to support non-preemptive thread services, which provide most of the functionality needed to hide remote latency. We also modified synchronization and communication services to function properly in the multi-threaded environment. Since CVM's architecture enforces the separation of the basic DSM services from protocol-specific functions, consistency models can usually be implemented without changing core CVM code. We were therefore able to restrict our changes to only a few lines of protocol consistency code.

The thread service uses a simple policy to decide when to switch between threads. We attempt to perform a thread switch whenever the current thread blocks on a remote request, or arrives at a barrier and is not the last local thread to arrive. Thread switches can also occur as the result of explicit application requests through a CVM API call.

We also modified CVM's core synchronization routines in order to reduce their communication requirements in multi-threaded environments. Barrier operations were modified so that all but the last local thread will thread switch upon arriving at a barrier. The last thread aggregates all local arrivals into a single per-node arrival message. Barrier release messages are handled similarly.

We extended this same idea to the application level in order to support reduction-like operations that would otherwise use global locks. A common pattern in parallel programs is to accumulate modifications to shared data structures locally, updating the shared structure only at the end of the current iteration. Transparently adding multi-threading to this type of application causes each local thread to update the shared data structure, resulting in additional (and unnecessary) synchronization and data messages. We added a *local barrier* mechanism that allows co-located threads to synchronize with each other. Such a mechanism can be used by the application to accumulate results from all local threads into a single remote update. Unfortunately, this type of mechanism cannot be generated automatically unless the reduction operations are already visible to the underlying DSM. CVM does support simple reduction types, but none of the applications in our study take advantage of them.

Additionally, the behaviors of both lock acquire and release operations have been changed. We implemented a local queue for each lock so that multiple local acquires result in only a single remote lock request. Threads that attempt to acquire a lock that has already been requested locally are placed on a local per-lock queue. The release code prefers the inhabitants of this queue over any remote thread, even if the remote thread requested the lock before the local threads. The result is neither fair nor guaranteed to make progress, but performs well for the applications in this study. This policy would probably need to be extended in order to efficiently support applications that use centralized work queues.

Although fine-grained thread systems can improve load-balancing by moving work to lightly-loaded nodes, our system implements coarse-grained, non-preemptive threads, and does not currently support thread migration.

4. Results

This section presents a detailed evaluation of MT's effects on a state-of-the-art multi-writer LRC paper, together a study of protocol-specific effects.

4.1 Experimental Environment

We ran CVM on a cluster of eight Alpha 2100 4/275 nodes. Each Alpha node has four 275 MHz Alpha processors and 256 Mbytes of memory. Experiments have shown between 10% and 20% performance degradation when using more than one processor per node, independent of whether MT is used. We therefore use only a single processor per node in order to avoid contention at the network interface, and all of our performance results are based on eight-processor runs. The Alphas run Digital UNIX V4.0, and are connected via Digital's GigaSwitch/ATM communications hub. Each node currently has a 155 Mb/s ATM interface.

CVM runs on UDP/IP over the ATM network. Simple 2-hop lock acquires take 937 µsecs, while 3-hop lock acquires take 1382 µsecs. Lock acquires are implemented by sending a request message to the lock manager, which then forwards the request on to the last requester of the same lock. This requires only two messages if the manager is also the last owner of the lock. Simple page faults across the network require an average of 1100 µsecs. Page fault times are highly dependent on the cost of mprotect calls, 49 µsecs, and the cost of handling signals at the user level, 98 µsecs. Minimal 8-processor barriers cost 2470 µsecs. Thread switches cost approximately 8 µsecs.

4.2 Application Suite

App	*Sync*	*Input*
Barnes	barrier	16K particles
Erle	barrier	Size = 125
FFT	barrier	64x64x64
Ocean	barrier, lock	258 x 258
Shpatial	barrier, lock	4K molecules
SOR	barrier	2048 x 2048
SWM	barrier	750 x 750
WaterNsq	barrier, lock	512 molecules
WaterSp	barrier, lock	4K molecules

Table 1: Application Specifics

Our application suite consists of nine applications: Barnes, Erle, FFT, Ocean, Shpatial, SOR, SWM, WaterNsq, and WaterSp. Table 1 lists specifics for the applications in our study. The *Sync* column indicates the synchronization operations used by the applications. All four applications that use locks were modified so that locks operations in per-thread reduction operations were merged at the node level before network communication takes place, as discussed in Section 2. All applications were also checked for problems related to the scope of global variables. Three applications, Barnes, WaterNsq, and WaterSp, had some of their global variables specialized for each thread. The last column shows the problem sizes used in our experiment.

Barnes is a modified version of the gravitational N-body simulation from Splash-2. Our version differs from the original version in that several small sections of code have been serialized in order to reduce synchronization. Erle computes variable derivatives using tri-diagonal solver. FFT is a 3-D fast Fourier transform that uses matrix transposition to reduce communication. Ocean is the contiguous ocean from Splash-2. It simulates large-scale ocean movements based on eddy and boundary currents. Shpatial is a modified version of the Splash-2 spatial water application that we obtained from the SHRIMP project [8]. The primary differences between Shpatial and WaterSp are that data locality has been increased and synchronization decreased. SWM is two dimensional stencil computation that applies finite-difference methods to solve shallow-water model. Erle and SWM were parallelized by the SUIF compiler [4] from sequential Fortran codes. We applied no advanced optimization techniques during compilation. Finally, WaterNsq and WaterSp are molecular dynamics simulations from Splash-2. While WaterNsq uses O(N^2) algorithms, WaterSp uses a more efficient algorithm that works by imposing a uniform 3-D grid of cells on the problem domain.

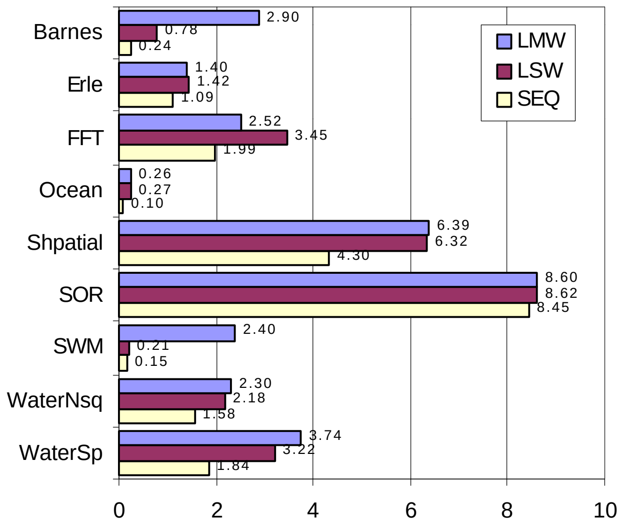

Figure 2: 8-CPU Speedups for LMW, LSW and SEQ

Figure 2 shows single-threaded eight-processor speedups of these applications with Lazy Multi-Writer (LMW), Lazy Single-Writer (LSW), and Sequential (SEQ) protocols. In general, the results with LSW and SEQ show smaller speedups than LMW, except FFT with LSW. The primary reasons behind LMW’s superior performance are better handling of false sharing, and the ability to make modify pages without distributed consensus. See Section 4.4 for more details. The expensive communication in this environment dictates relatively poor overall speedups. However, the range of performance shown in Figure 2 provides an opportunity to evaluate MT in several different contexts.

We use the term “multi-processor speedup” to refer to speedup of eight-processor runs over the single-processor case. “Multi-thread speedup” refers to the speedup versus the single-threaded eight-processor case. Unless otherwise qualified, “speedup” will refer to this latter definition throughout the rest of this paper.

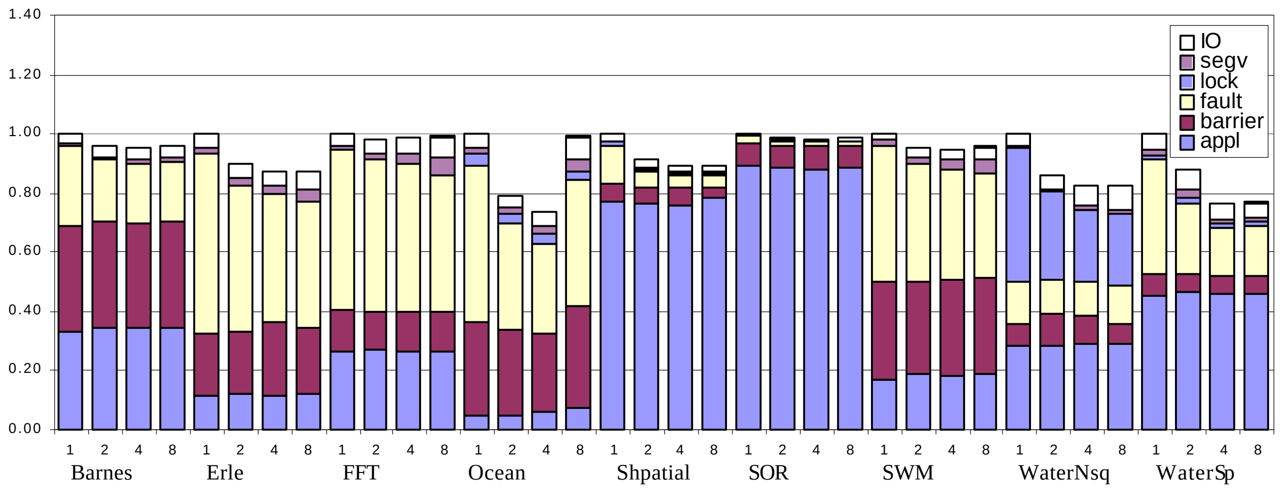

Figure 3: Normalized LMW Execution Time

4.3 LMW Performance

Figure 3 shows LMW's performance for one, two, four and eight threads per node, normalized to single-threaded execution times. We restricted the number of threads per node to be a power of two in our experiments because some of our applications expect this.

Table 2 shows details of multi-threading's effect on the low-level behavior of LMW. The table gives total bandwidth consumption, details of remote page and lock requests, and information on diff creation and use. *Remote Faults* and *Remote Locks* list the total number of faults and lock acquisitions that require network communication. *Overlapping Faults* and *Locks* give measures of how effective the system is at overlapping multiple remote accesses. These numbers are counts of how many remote requests are currently outstanding each time another remote request is initiated. *Contention Faults* and *Locks* measure contention for resources between local threads by counting the number of times multiple threads blocked on the same page or lock.

Overall, the results are quite promising. Ocean, WaterNsq and WaterSp each achieved multi-thread speedups of at least 18% with four threads per node. Erle and Shpatial also improved by at least 10%, and Barnes and SWM by at least 5%. The final two applications, FFT and SOR, sped up by only 2%. Since SOR's speedup is near-linear even in the single-thread case, we did not expect it to improve significantly. We've included it primarily to show that our multithreaded implementation imposes little additional overhead, even when there is very little remote latency to hide.

Figure 3 also breaks down normalized execution times into contributions from application time (which includes all time spent executing local DSM protocol code), time spent waiting at barriers, time spent waiting on faults, time spent waiting for locks, IO handling time, and time spent handling segmentation violations. IO handling time is the time spent servicing incoming barrier, lock and remote fault requests. Segmentation violation (segv) handling time measures the overhead of the operating system upcall to user space when invalid accesses to shared memory occur. Invalid accesses are used to inform the DSM when consistency actions need to occur. Although we can not measure this cost directly, we can get an approximate value by multiplying the number of segmentation violations by an average cost of 98 µsecs per violation. Application times vary slightly with different numbers of threads because we are unable to exclude the influence of other factors that are affected by multi-threading, such as cache and TLB behavior.

The primary advantage of MT is that time spent waiting for the completion of remote requests can be used by other local threads. As expected, MT reduced fault and lock times by hiding remote latencies with useful local work. Overall, fault times reduced by an average of about 35% for both four and eight threads, indicating that either most latency is hidden at four threads (and the remain latency is software overhead), or fours thread are sufficient to exploit all available parallelism.

Only one of the applications, WaterNsq, has significant lock delay. MT addresses all lock delays effectively, with improvements in lock handling times continuing to increase until eight threads for all of the applications that used locks. Lock waiting time improves by an average of 20% at four threads, and more than 44% at eight threads. The *Remote Lock* column in Table 2 shows that there is essentially no change in the number of remote lock acquisitions as the degree of multithreading increases. This implies that we are able to successfully aggregate all local thread locks for a given reduction into a single remote lock access. This conclusion is supported by the lack of local lock contention.

These gains are somewhat offset by the fact that the time

Appl	*T*	*BW (kbytes)*	*Remote*		*Overlapping*		*Contention*		*Diffs*	
			Fault	*Lock*	*Faults*	*Locks*	*Page*	*Lock*	*Create*	*Use*
Barnes	1	25318	4117	0	0	0	0	0	2563	13879
	2	25327	4102	0	3905	0	325	0	2584	13893
	4	25347	4082	0	10357	0	1054	0	2627	13921
	8	25403	4087	0	20251	0	2392	0	2713	14005
Erle	1	103368	11809	0	0	0	0	0	10996	18379
	2	103143	9842	0	12424	0	3759	0	10948	18333
	4	103267	9153	0	26055	0	11623	0	10981	18369
	8	103316	9140	0	43940	0	24096	0	10991	18381
FFT	1	61191	6002	0	0	0	0	0	6016	7070
	2	61204	5924	0	1086	0	5913	0	6016	7070
	4	61223	5913	0	3230	0	17826	0	6016	7070
	8	61263	5913	0	6713	0	41538	0	6016	7070
Ocean	1	118350	21237	641	0	0	0	0	13840	26424
	2	102609	18736	639	14873	0	1047	0	10772	25331
	4	79701	18082	648	71877	0	4260	0	13161	24307
	8	138212	27168	550	25091	0	11972	0	20590	37613
Shpatial	1	32754	7254	47	0	0	0	0	1076	7532
	2	32755	7217	48	5967	0	1266	0	1076	7532
	4	23419	5121	48	5409	0	5224	0	1075	5477
	8	23419	5012	48	11372	0	3059	0	1076	5484
SOR	1	6736	1092	0	0	0	0	0	1091	1092
	2	6741	624	0	936	0	0	0	1092	1092
	4	6741	625	0	934	0	0	0	1092	1092
	8	6741	624	0	936	0	0	0	1092	1092
SWM	1	62975	19632	0	0	0	0	0	15193	42399
	2	62954	17232	0	16080	0	7772	0	15112	42318
	4	62957	17047	0	21651	0	23334	0	15131	42343
	8	62957	16932	0	31657	0	54439	0	15126	42338
Water Nsq	1	22199	3679	12480	0	0	0	0	1890	8551
	2	22514	4120	12479	2532	12239	510	0	2771	12186
	4	22766	4100	12479	4682	36575	3204	0	3365	14642
	8	23164	3900	12479	8813	84665	6374	0	3933	16963
WaterSp	1	38222	57437	48	0	0	0	0	8244	57708
	2	38232	57404	48	36505	0	22832	0	8243	57701
	4	27369	41035	48	12973	0	485	0	8244	41324
	8	27357	40882	47	23616	0	3904	0	8244	41324

Table 2: LMW: DSM Actions

spent waiting at barriers generally increases slightly as more threads are used. The reason is that MT introduces more variability into the system by breaking a single node's work into thread-sized pieces, and interleaving the performance of those pieces in a non-deterministic manner. Additionally, there is a great deal of variation among processors in how successful they are at hiding remote latency. MT can not be used to hide barrier-waiting time because barrier arrival messages are not sent until all local threads have arrived.

Directly measuring the overlap of communication and computation is difficult because we have no way of determining exactly when replies arrive in our system. However, the number of overlapping requests gives us some idea of how effective we are at finding parallelism. Each incident of two overlapped requests means that we have at least some chance to completely hide the remote latency of one of them. Increasing amounts of local contention, on the other hand, mean that we are not able to find enough parallelism to keep all local threads busy.

For example, if thread T_1 blocks on a remote page fault, the system switches to T_2, and then thread T_2 blocks on either a remote lock or remote fault, *Overlapped Faults* will be incremented. Likewise, if T_2 had blocked on the same page as T_1, then the amount of local contention for pages would be incremented. In almost all cases, the number of overlapped faults *and* the amount of local contention increase uniformly as the level of multi-threading increases.

SOR is one of the exceptions. The number of overlapped faults stays constant across two through eight threads per node for SOR because each barrier epoch has only two remote page faults. Hence, all available "request" parallelism is exploited with two threads.

Both versions of Spatial have decreasing amounts of page contention as the number of threads increases. We speculate that increasing the number of threads decreases the incidence of false sharing between them. The regions of shared data accessed by local threads are likely to be further apart with more threads, and therefore less likely to be on the same page.

Neither Ocean nor Spatial has any overlapping lock requests. These applications use small numbers of locks, and the lock responses are fast. However, the primary reason for the lack of overlap is probably that almost all lock acquisitions are used in support of reduction operations. Hence, our code modifications have already ensured that all local threads combine to make just one remote lock acquisition.

To some extent, local contention gives one measure of how well suited an application is for transparent multi-threading. SWM, for example, generated approximately 17,000 remote faults, and has local page contention approximately 7,000 times per additional local thread. The obvious implication is that 7,000 of the original 17,000 pages are going to be accessed by every local thread. However, this measure of contention does not necessarily mean that no overlap was accomplished, as it gives no notion of how much computation was performed by the second thread before it also blocked.

Nonetheless, large amounts of contention in an application indicate that it is unlikely to benefit from naively increasing the level of multi-threading. Such applications often need source modification in order to fully exploit MT's potential.

The last two columns of Table 2 show the number of diffs created and applied. Recall that diffs are used to summarize modifications to a single virtual memory page. One of the performance problems that MT can introduce in a multi-writer protocol is an increased number of diffs. Multi-threading may break the modification of a given page into modifications by different threads. Per-thread diffs may be created instead of a single combined diff if the threads are on different nodes, or modify their portions of the data at different times. The *Diffs Create* and *Diffs Use* columns of Table 2 show that this problem is negligible for Erle, FFT, SOR and SWM. Moreover, it is only a minor problem for Barnes as only approximately 10% more diffs are created in the eight thread case. Shpatial and WaterSp actually use fewer diffs as the number of threads increases, although the same number are created. This situation shows that increasing the number of threads is actually increasing locality, either in time or in space. On the other hand, Ocean generates 48% more and WaterNsq creates 108% more diffs with eight threads than with one, indicating that locality is being decreased.

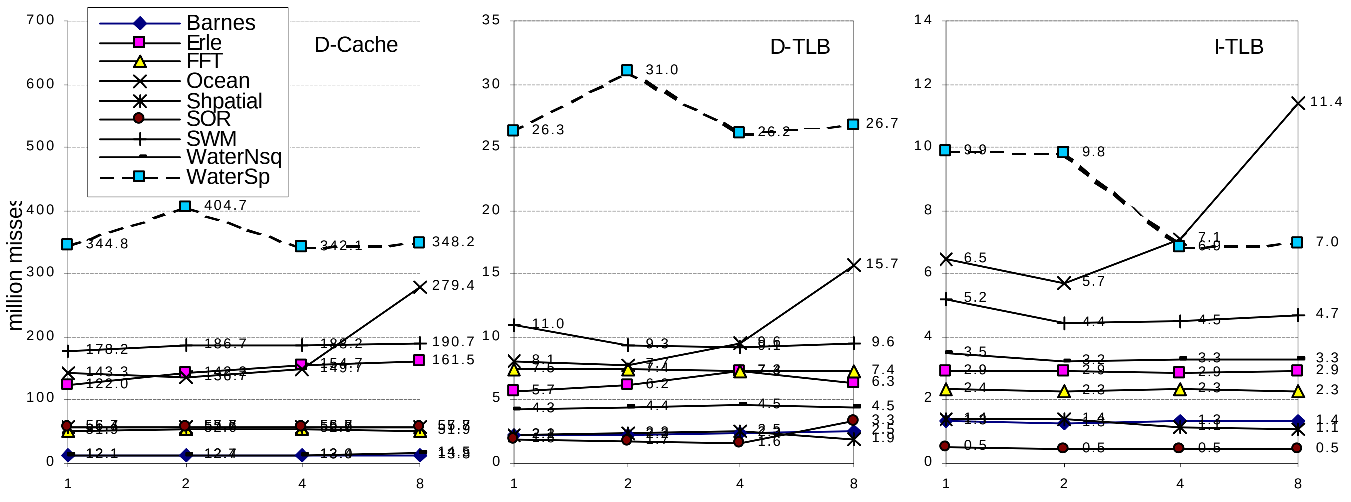

Figure 4: Cache and TLB Statistics for Eight Threads

The number of messages carrying diff remains constant for five of the nine applications, but increases with more threads for both WaterNsq and Ocean. By contrast, the number of remote page faults decreases by like amounts for the two versions of Spatial. As above, these differences probably reflect changes in locality induced by new interleavings of shared accesses as the number of threads increases.

Breaking a single diff into multiple diffs can increase the total size of created diffs. For example, if a single-threaded application modifies the same region of shared memory multiple times, increasing the level of multi-threading may result in each modification being summarized in a separate diff. All diffs but the last are pure overhead because only the final result needs to be seen at other nodes. However, Table 2 shows that the numbers of remote faults in Ocean and WaterNsq goes up quickly, but that the total amount of communicated data changes only slightly. We therefore infer that the bulk of the diffs created by subdividing single-thread diffs are non-overlapping.

4.3.1 Effects on the Memory Hierarchy

The decrease in locality caused by MT is potentially present not only at the page level, but also in caches and TLBs. Figure 4 shows the total number of misses in the data cache (D-cache), the data translation look-up table (D-TLB) and instruction translation look-up table (I-TLB). These numbers refer to runs on an eight-node SP-2 rather than the alpha machines because we currently have no way of getting the corresponding numbers on our alpha machines. Multi-threading speedups on the SP2 were qualitatively similar to those on the Alphas, but lower. The results are not directly comparable, as the machines differ in many architectural respects, including processor, network, and cache configuration. The Alphas and the SP-2 also differ in virtual memory page size. We partially compensated for this by forcing the SP-2 version of CVM to use the Alpha's 8 KByte page size as the unit of shared coherence. The SP-2 has only 64 Kbytes of cache per processor, while each Alpha processor has 16 Kbytes in the first-level cache, and 4 Mbytes in the second level. Hence the cache effects shown Figure 4 are probably more pronounced than on the Alpha cluster.

Although there is a great deal of variation among the applications, both cache and TLB misses generally increase with the level of multi-threading. The two outliers are Ocean, which shows significant degradation of locality with increasing number of threads, and WaterSp, which has fewer TLB misses at four threads than at one. Ocean's poor locality is caused by the large number of thread switches. WaterSp's good locality is probably due to the decreased message traffic and consequent decrease in operating system calls.

Part of the reason that cache and TLB hit ratios do not seriously degrade is that our threads are reasonably coarse-grained. Threads in CVM are non-preemptive, implying that thread switches can occur infrequently. Moreover, DSM systems like CVM makes system calls at a high rate, and probably have relatively low hit ratios as a consequence.

4.4 Multi-threading with LSW and SEQ Protocols

We ran the same applications in our environment with two additional memory consistency protocols in order to separate protocol-specific interactions with MT from more general effects. The additional protocols are LSW, and SEQ. Complete details can be found in Keleher [1]. Briefly, however, LSW differs from LMW in that it does not allow multiple concurrent writers, and therefore transfers data in complete pages rather than by using diffs. False sharing is tolerated similarly in both protocols, but true sharing causes LSW's performance to degrade sharply. SEQ is also a single-writer protocol, but implements sequential consistency rather than the relaxed consistency model supported by the other protocols. The comparative speedups in Figure 2 show that LMW performs the best, followed by LSW, and then by SEQ. With LSW and SEQ, Barnes and SWM were actually slowed down

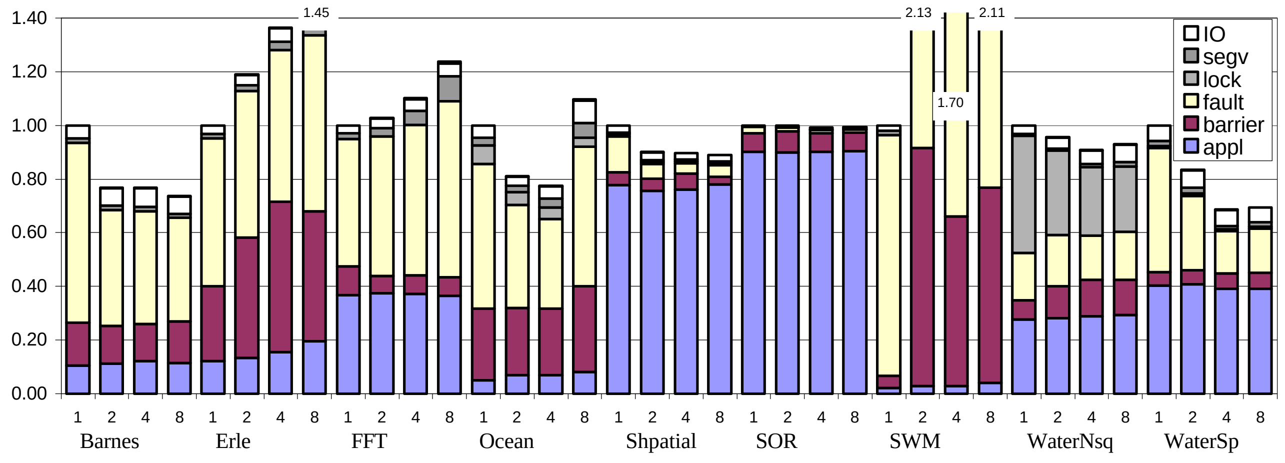

Figure 5: Normalized LSW Execution Time

by factor of three or more. Ocean again has no speedup at all. LSW FFT is actually faster than LMW FFT because it does not create diffs, and FFT modifies a large amount of data.

Neither LSW nor SEQ performs as well as LMW in the single-threaded case because of increased remote request times. We expected MT speedup to increase correspondingly. However, this turns out not to be the case.

4.4.1 Multi-Threading and LSW

Figure 5 shows the performance of one, two, four and eight threads per node for LSW. The elapsed times of all multi-thread runs were normalized to the single thread execution times. Figure 5 also breaks elapsed times down into the same categories as Figure 3 did for LMW. Table 3 shows detailed results for both LSW and SEQ, the protocol discussed in the next section. Looking only at the LSW results for now, the columns show bandwidth requirements, counts of remote faults, lock messages, and incidents of page contention.

Overall, the results are somewhat disappointing. Although Barnes, Ocean and WaterSp sped up by more than 20%, Erle, FFT and SWM all performed markedly worse than their single thread counterparts. The primary problems for Erle and SWM is increased load imbalance, as barrier wait times increased markedly. As mentioned above, the performance of LSW is relatively unstable in the presence of true sharing because processors contending for the same page can result in it ping-ponging across the network (page thrashing). We speculate that this caused the load imbalance.

FFT’s poor performance, on the other hand, is due directly to increasing fault times. Table 3 shows that the eight-threaded run has 8324 remote faults and 41245 incidents of local contention. This implies that for every thread that sends a remote page request, five other local threads block on the same page before the reply returns. This explains why little improvement is seen. However, the reason why performance decreases with more threads is that more remote faults occur (20% more for eight threads than for one), and bandwidth requirements are correspondingly higher (23%). We can only speculate that the underlying reason for the increased page faults is decreased locality.

Shpatial and WaterNsq sped up by 10% and 9% with four threads per node, respectively. SOR sped up by only 1%, but, again, this application has little request parallelism to exploit. Multithreading effects can still be observed in these applications, i.e. lock and fault time decreased.

Overall, six applications, Barnes, Ocean, Shpatial, SOR, WaterNsq, and WaterSp had similar MT speedups in both

Apps	T	LSW				SEQ			
		BW KBytes	Remote Faults	Lock Msgs	Page Cont.	BW KBytes	Lock Msgs	Remote Faults	Page Cont.
Barnes	1	380123	47385	0	0	1083905	0	133328	0
	2	361590	45077	0	812	1145333	0	140905	7215
	4	345844	43117	0	2280	1361077	0	167437	19720
	8	355526	44288	0	5289	1504527	0	185069	36046
Erle	1	111889	18400	0	0	126048	0	15498	0
	2	145359	22505	0	3443	140357	0	17260	4154
	4	151156	23179	0	11451	177209	0	21799	12989
	8	154597	23577	0	24938	193089	0	23751	29934
FFT	1	49070	6946	0	0	99548	0	12293	0
	2	50743	7151	0	5856	100715	0	12437	5818
	4	54000	7548	0	17614	102302	0	12633	17448
	8	60224	8324	0	41245	105477	0	13025	40854
Ocean	1	171202	21368	1121	0	382394	1118	47156	0
	2	154463	19194	1078	1099	397941	1114	49048	921
	4	178257	22173	1133	4590	446296	1004	55021	3164
	8	309711	38591	1134	14008	682274	932	84116	10004
Shpatial	1	58623	7302	85	0	117290	82	14442	0
	2	58630	7302	84	1258	116176	81	14301	3930
	4	42114	5254	85	5264	179979	84	22180	7478
	8	42116	5253	85	3095	239043	83	29477	7820
SOR	1	9127	1092	0	0	17754	0	2184	0
	2	9135	1092	0	0	17754	0	2184	0
	4	9135	1092	0	0	17754	0	2184	0
	8	9135	1092	0	0	17754	0	2184	0
SWM	1	2345351	296710	0	0	2881982	0	355500	0
	2	3931271	490373	0	40828	3247003	0	400541	89356
	4	3615995	451211	0	114669	3006180	0	370835	159952
	8	3440148	429426	0	168255	2759358	0	340394	221365
Water Nsq	1	33952	5037	21840	0	51394	21839	6209	0
	2	51310	7118	21839	765	68635	21838	8330	1021
	4	50951	7076	21837	3649	63097	21837	7647	4201
	8	61692	8290	21838	7029	74916	21763	9103	8676
WaterSp	1	463863	57482	81	0	630753	83	77555	0
	2	463930	57485	81	23209	672451	84	82699	21665
	4	331865	41102	84	397	650425	84	80099	5619
	8	331862	41103	84	3852	833277	81	102685	7720

Table 3: LSW and SEQ Behavior

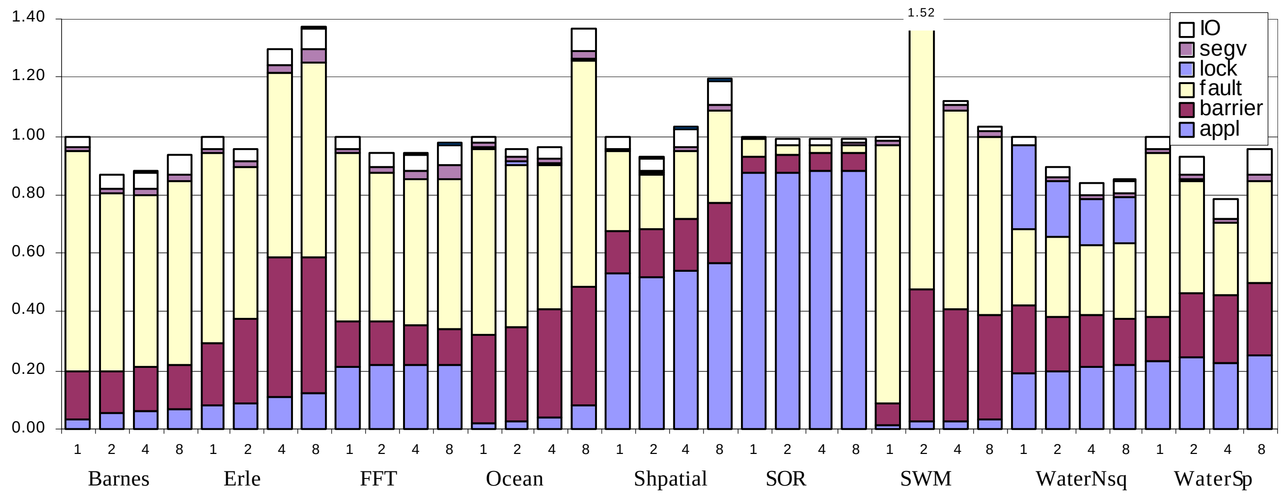

Figure 6: Normalized SEQ Execution Time

LMW and LSW. Three applications, Barnes, WaterNsq, and WaterSp, have lower speedups with LSW. This is somewhat surprising, because LSW's lower processor speedups imply that there is more potential for MT speedup. Moreover, while remote miss latency accounts for an average of 34% of single-thread application time under LMW, the corresponding number for LSW is 45%. However, while the LMW number drops to 24% at four threads, miss latency drops to only 36% for LSW.

Part of the problem is the effect of MT on LSW. Recall that LMW handles true sharing without network communication, whereas multiple processors attempting to access a common page under single-writer protocols might cause page thrashing (and hence additional remote misses). While the average number of remote faults at eight threads decreases by 12% for applications under LMW, the corresponding number of faults *increases* by an average of 27% under LSW. This additional, non-deterministic work increases barrier imbalance, and makes it unlikely that significant performance gains will be achieved

The last LSW column in Table 3 shows that the amount of local page contention under LSW is similar to the contention observed under LMW.

4.4.2 Multi-Threading and SEQ

Figure 6 shows the performance of the applications under SEQ, our sequentially-consistent single-writer protocol. As before, we show execution times for one, two, four and eight threads per node, normalized to single-thread times. These execution times are broken down into the same categories as before. The last four columns of Table 3 show bandwidth consumption, the number of lock messages, remote page faults, and page contention for SEQ.

Overall, the performance of MT for applications running on SEQ shows the same trend as that of LSW, only more so. Despite having lower processor speedups than either of the other protocols, and hence more opportunity for improvement, MT improves performance by 20% for only one application, WaterNsq, and by at least 10% for only two more. Three of the applications do significantly worse with MT. As with LSW, the performance degradation results from a combination of increase fault times, and increasing barrier imbalance.

Fault latency averages more than 52% of the single-threaded execution cost, and 40% even for the best MT case. The number of remote faults at eight threads increases by even more than with LSW, by an average of 39% across all of our applications. Moreover, while the incidence of local page contention is similar for most applications under both LMW and LSW, contention is much more pronounced under SEQ. These effects are likely due to the fact that less page thrashing occurs under LSW because the protocol is able to hide a great deal of false sharing by delaying consistency actions. By contrast, consistency actions are performed immediately with SEQ. Bandwidth requirements increase correspondingly.

In contrast to LSW, and especially LMW, eight threads perform better than four threads for only a single application, SWM. The number of remote page faults and the amount of consequent fault latency explain the slowdown for most of the applications. SWM's processor speedup under SEQ is so poor that it is difficult to understand its performance.

Finally, note that the bandwidth requirements for SEQ are much higher than for the other protocols. High bandwidth tends to hurt MT speedup because data copying can not be overlapped with any other local activity.

4.5 Useful and Non-Useful Thread Switches

The notions of overlapped requests and local lock and page contention discussed in Sections 4.3 and 4.4 give some idea of the effectiveness MT and the amount of parallelism that it can exploit. However, these metrics are incomplete in that they don't give any indication of the amount of computation performed before threads block. For example, these statistics will count situations where t_1 blocks on a page request, the system switches to t_2, which computes a while, and then blocks on a request for a different page. However, they give no idea of how long t_2 computes before the reply for t_1's request returns. Unfortunately, this number would be difficult to obtain in our environment because replies are not signaled by interrupts. Our system can not know that a reply has returned until the currently active thread blocks (for whatever reason) and we check for incoming messages prior to scheduling a new thread.

However, we can easily instrument our system to check the state of other threads each time the active thread blocks on a remote request or barrier arrival, and also gives us a better idea of how much remote latency is overlapped with useful work. We can also get a rough estimate of the major causes of useful work not being available.

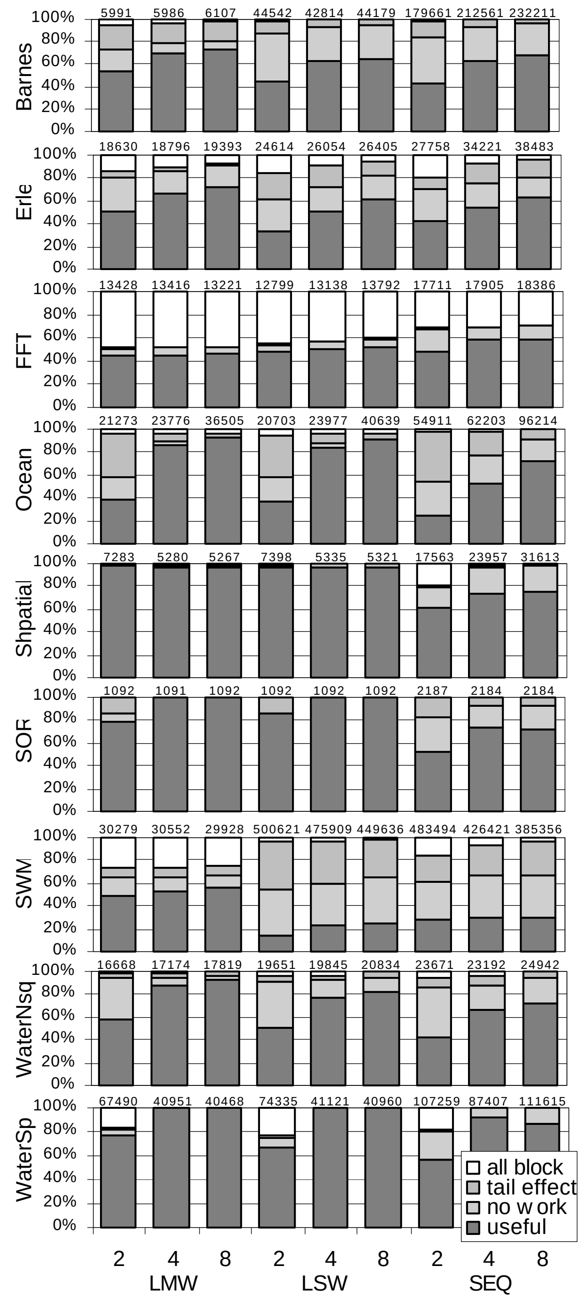

Figure 7: Useful and Non-Useful Thread Switches

Figure 7 divides such thread situations into four different categories for two, four, and eight threads and all three of our protocols. "Useful" means that useful work is available when a thread switch occurs, either because at least one thread has not left the last barrier, or because a reply has arrived for at least one blocked thread. The other three categories all represent situations where no other thread is ready to run, but for different reasons. There are far too many possible scenarios to be enumerated in a single chart when each node has up to eight threads. Instead, we distinguish just two special cases. The first is "all block", which means that all other threads are blocked on the same page. MT has no chance of improving performance in such situations. The second is "Tail Effect", which refers to the situation when all other threads have already arrived at the next barrier. Increasing the level of MT *can* improve performance if the last thread to arrive at a barrier incurs multiple faults after all other threads have arrived at the next barrier. Additional threads might cause these "tail" faults to be distributed across multiple local threads, allowing the data requests to proceed in parallel. Finally, "no work" refers to all other cases where no work can be found. The number at the top of each bar is the total number of remote requests and barrier arrivals at which we attempt to find a thread with useful work.

Looking first at the eight-thread numbers for LMW, useful work is available at half or more of thread switches for all applications except FFT, where useful work is available only 45% of the time. Over 90% of thread switches are useful for five of the nine applications, and 70% for two more. For FFT and SWM, the two applications that have the smallest percentage of useful thread switches, the majority result from all threads being blocked on the same page. Not coincidentally, these applications are among the poorest in terms of MT speedup. These applications would not benefit from increased multi-threading, and in fact Figure 3 shows that both do better with four threads than eight.

Across all protocols, the percentage of "tail effect" switches tends to decrease with increasing MT level, and the percentage of useful switches tends to increase. The reason is that increasing numbers of local threads cause more thread switches to occur at each barrier. All switches to threads that are just leaving barriers are useful.

Both the "tail effect" and "no work" switches occur more frequently for LSW and SEQ than for LMW. It is likely that many of the threads at "no work" switches have also already arrived at the next barrier, just as all threads have done with "tail effect" switches. The percentage of such switches increase for the single-writer protocols because poor handling of false sharing tends to increase the variability in finishing rates between local threads, i.e. some threads get delayed by page thrashing more than others. Hence, the number of page faults that occur after all other threads arrive at the next barrier goes up.

5. Conclusions

This paper has presented the results of our experiments in latency-hiding via per-node multi-threading. Three of our applications sped up by at least 17% under our multi-writer LRC protocol, and all gained at least some benefit from the multi-threading. We identify the following as limiting factors to multi-thread speedup:

Lack of Parallelism
The "useful" portion of the bars in Figure 7 shows the percentage of time that work is available when a thread switch occurs. The extent to which this percentage continues to rise as the number of threads per node increases gives some indication about the amount of available parallelism. However, much like runtime data-race detection methods, this information does not give any upper bound on parallelism. It shows only the parallelism that our runtime system was able to exploit.

Local Contention for Resources
The threads on a single node often contend or block and wait for the same resource. Any instance of multiple local threads waiting on the same resource means that multi-threading potential is being wasted.

Increasing the level of multi-threading can sometimes overcome contention induced by false sharing. However, contention introduced by true sharing can usually only be addressed through source modification. In other words, if all local threads contend for the same resources, no amount of multi-threading will improve performance of this part of the code. In fact, care needs to be taken to ensure that performance is not degraded. The reduction operations discussed below are one example of such true sharing.

Reduction operations
As the single-thread-per-node model is nearly universal, programmers tend to accumulate results locally before communicating with remote threads. These operations are essentially reductions. Naively splitting a single thread into multiple threads can result in each thread individually communicating local results to the same remote location.

Reduction operations should ideally be identified in the source. The result would be better performance in both the single-threaded case and similar performance in the multi-threaded case. Since the reductions were not identified in the source of our applications, we modified the source to take advantage of CVM-provided local barriers. The contributions of all local threads are then aggregated into a single remote request.

Load Imbalance
Unequal distribution of load across nodes can also prevent MT from speeding up the application. All of our applications start with uniform load distributions, aside from small serial portions. However, DSM communication can throw this balance off by delaying some threads and processors more than others.

Caches and TLBs
Context-switching between threads reduces the chance that caches and TLBs will retain state for a given thread by the time it switches back in. The performance of even local computation is then degraded by increased cache and TLB misses. A thread scheduler might attempt to increase locality by using an approach closer to LIFO than FIFO. Our scheduler does not currently take locality into account.

Application perturbation
Multi-threading changes the order that events occur, both within and between nodes. This non-deterministic effect can either improve or reduce different aspects of application performance. However, to the extent that this perturbation increases performance variability between different nodes, it hurts overall performance.

Thread switch cost
Although not as expensive as remote accesses, switching between local threads has a significant cost and can be a factor with enough threads. One approach to reducing this cost and to increasing our coverage of existing parallelism is to combine a lightweight, fine-grained threading package with adaptive load-balancing [9]. Lightweight thread packages [10] can be fine-grained enough that it is possible to load-balance through thread migration, and to minimize unhealthy interactions with the underlying DSM by bin scheduling of threads [11]. However, such systems usually do not allow threads to be blocked, i.e. all threads are *run-to-completion*. The challenge is to build a lightweight threading system without changing the programming model in this way.

The primary goal of this paper was to evaluate the effect of MT on DSM performance, and to identify factors that limit its performance benefits. We found that MT improved the performance of all of our applications under our primary protocol, two by more than 20%. A number of limitations are discussed above.

The second goal was to see if MT could be added without application modification. Our applications are all parameterized by command-line options to handle different numbers of nodes. Hence, the system can usually add per-node MT transparently to the application without affecting correctness. However, MT may not be transparent to application performance. In fact, we found that modifications were generally needed in order to prevent lock-based reductions from causing network communication for each local thread. No matter how cheap thread operations are made, excess communications would hurt performance. These modifications would not have been necessary if our applications had been written for an API that explicitly calls out reduction operations (i.e. PVM [12]or MPI [13]).

Reductions are relatively easy to recognize in the source. However, the more general problem of determining whether increased multi-threading will improve performance is much more difficult. Determining whether the *potential* for improvement exists is relatively easy. Applications that spend significant amounts of time waiting for remote requests to be served have potential for improvement. All of our applications except SOR fit this category. However, two properties must hold in order for this potential to be realized.

First, increased MT must not change the interleaving of local threads to the extent that more network faults occur. This can happen when a thread that accesses an object more than once is split into multiple local threads, such that the accesses get distributed among the resulting threads. If the threads are scheduled so that the accesses occur far apart, interaction with other nodes might mean that the resulting resource needs to be fetched multiple times, instead of the one time needed by the original thread. This condition is very difficult to identify. Moreover, it is a property not just of the application, but also of the application's environment.

Second, there is no opportunity for parallelism of data requests if each thread's performance is dominated by accesses to essentially the same set of resources (pages or locks). For example, consider the breakdown of scheduling attempts for eight threads under LMW in Figure 7. A large number of such attempts for both FFT and SWM result in the "all block" situation, meaning that all threads are blocked waiting for the same resource. Increased MT will only add overhead to such applications because additional threads will just block on the same resource. Not surprisingly, Figure 3 shows that both applications perform worse at eight threads than at four. This condition is relatively easy to detect through post-mortum analysis.

Our final goal was to evaluate the influence of the type of protocol on MT's speedups. We measured the effect of MT on two other protocols: LSW, a single-writer LRC protocol, and SEQ, a single-writer sequentially consistent protocol. In general, we found that MT realized much less of its potential with the latter two protocols than with LMW. Two aspects of the protocols dictate this difference: communication requirements and the handling of false sharing.

Both of the single-writer protocols require large amounts of bandwidth because they move data as complete pages rather than as diffs. Furthermore, the page thrashing that occurs with concurrent accesses to the same page increases the number of network page faults. While MT may merge previously separate updates, such behavior is entirely random and hurts performance just as often as it helps. The higher bandwidth hurts MT by increasing the software overhead incurred by communication primitives. Such operations limit MT speedup because they can not be overlapped with other local computations.

LMW was carefully designed to address all forms of sharing. Such care is crucial because the page thrashing that can result from sharing in single-writer protocols can greatly hurt performance. Moreover, false sharing is more likely to happen in CVM-like DSMs than hardware shared memory machines because the base coherence granularity is much larger. Virtual memory pages are 8192 bytes or larger on most new systems.

The single-writer protocols used in this study address page thrashing by freezing newly arrived pages for a short period before allowing them to be invalidated. While this can reduce page thrashing, it also tends to decrease performance for applications that synchronize at fine granularities.

We might wonder about the applicability of these techniques to other environments, specifically environments with lower-cost communication. For example, the performance of our communication primitives is poor compared to state-of-the-art ATM implementations, Myrinet [14], or Memory Channels [15].

While it might seem that MT would be less useful in such environments, the reality is likely the opposite. As noted back in Section 1, the only portion of request latencies that can be used locally by MT is wire time and remote computation. With bandwidth in the half-a-gigabyte range and one-way latencies of less than 10 *u*secs, wire time is likely to be insignificant for applications that are not bandwidth-limited, as ours are not. Systems developers are therefore likely to use low-cost polling rather than much more expensive signals to detect incoming messages. The disadvantage of polling is that incoming requests might not be detected in a timely manner if communication is not regular. Hence, the "remote computation" seen in such a system is likely to be quite large, implying that MT will remain a useful technique.

References

[1] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architec-

ture and Performance," in *Proceedings of the 22th International Conference on Computer Architecture*, May 1995.

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

[4] R. P. Wilson, R. S. French, C. S. Wilson, J. M. Amarasinghe, S. W. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, pp. 31-37, December 1994.

[5] W. Yu, C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, pp. 18--28, February 1996.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

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

[8] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[9] A. Itzkovitz, A. Schuster, and L. Wolfovich, "Thread Migration and its Applications in Distributed Shared Memory Systems," Technion IIT LPCR #9603, July 1996.

[10] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews, "Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations," in *Proc. of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[11] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread Scheduling for Cache Locality," in *Proceedings of the 7th International Conference on Architectural Supports for Programming Languages and Operating Systems*, 1996.

[12] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," in *Concurrency:Practice and Experience*, December 1990.

[13] "MPI: A Message-Passing Interface," 1994.

[14] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, pp. 29-36, 1995.

[15] R. Gillett, "Memory Channel Network for PCI," *IEEE Micro*, vol. 16, pp. 12-18, 1996.