Multi-threading and Remote Latency in Software DSMs

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 a software DSM. As with hardware systems, multi-threading in software systems can be used to reduce the costs of remote requests by switching threads when the current thread blocks.

We added multi-threading to the CVM software DSM and evaluated its impact on performance for a suite of common shared memory programs. Multi-threading resulted in speed improvements of at least 17% in three of the seven applications in our suite, and lesser improvements in the other applications. However, we found that i) good performance is not always achievable transparently for non-trivial applications, ii) multi-threading can negatively interact with DSM operations, iii) multi-threading decreases cache and TLB locality, and iv) any multi-threading speedup is dependent on available work.

1. Introduction

This paper presents an empirical evaluation of the use of per-node multi-threading to hide remote latencies in CVM [1], a 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. The primary drawback is that DSMs usually 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. Hence, this time is wasted at the local processor.

The primary performance advantage of per-node multi-threading is that multiple threads can ensure that work is available when the currently active thread stalls on a remote request. If the level of multi-threading is high-enough, all latency other than local OS overhead can be overlapped with useful local computation. 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. So-called *fine-grain* programs have several advantages, including architecture independence, clarity of expression, implicit load balancing, and ease of code generation for parallelizing compilers [4].

Three of the seven applications sped up by at least 17%, and the others by lesser amounts. However, our performance does not come close to the potential speedup implied by processor utilizations [5]. We have identified five contributors to this shortfall, and present strategies for dealing with them, if applicable:

1. 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.
   There are essentially two strategies for dealing with local contention. First, one could create a large enough number of threads that all of them are never blocked at the same time. This is too expensive in our environment, but is practical in systems where threads are very lightweight [6]. The second approach is source modification. We used this approach in some of our applications.
2. Reduction operations - As the single-thread-per-node model is nearly universal, programmers tend to accumulate results locally before communicating with other threads whenever possible. These operations are essentially reductions. When the single thread is split into multiple threads, each communicates local results to remote threads, resulting in extra communication and working counter to the programmer's optimizations.
   Reduction operations ideally should 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.
3. 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 memory-system aware thread scheduler would use an

approach closer to LIFO than FIFO. Our scheduler does not make this optimization.

4. Application perturbation - Multi-threading changes the order that events occur, both within and between nodes. This has a non-deterministic effect on performance.
5. Thread switch cost - Although not as expensive as remote accesses, switching between local threads does have a significant cost.

One of our primary goals was to see if multi-threading could be added without modifying applications. Our applications are all parameterized by command-line options to handle different numbers of nodes. Hence, the system can usually add per-node multi-threading transparently to the application without affecting correctness. However, as discussed above, multi-threading may not be transparent to application performance.

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, and Section 4 describes our performance. Finally, we conclude in Section 5.

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. Systems consist of a number of threads that can transparently access one or more shared segments. 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 simplicity. More than a single thread per node would introduce the frictional costs implied by thread switching. A single thread per node simplifies handling the scope of heap and stack-allocated data. This data is usually private to each thread; only designated, shared segments are visible to more than a single processor. The single-threaded scheme also has the advantage of making the relationships between threads uniform. Sharing between any pair of threads costs the same as any other pair. Hence, the programmer or compiler that distributes the data need only to accommodate a single level of sharing.

Multi-threaded nodes add an additional level to the hierarchy of memory access times, i.e. threads that are co-located 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, that data 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.

Although the above problem affects performance, it does not affect correctness. However, the most immediate consequence of per-node multi-threading is that whether two threads see the same instances of global program data depends on whether they are co-located on the same processor. We address this discrepancy by disallowing modifications of 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.

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 TreadMarks [7], CVM is written entirely as a user-level library and runs on most UNIX-like systems. Unlike TreadMarks, 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*[8]. 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) [9] 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.

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. The `SIGSEGV` signal handler examines local information 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. When 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. Non-preemptive threads provide all the functionality needed to hide remote latency. We also modified synchronization and communication services to function properly in multi-threaded environments. 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 consistency protocol code.

The thread services uses a simple policy for scheduling. Thread switches always takes place when remote requests are sent, and when *misplaced* replies are received. A reply is misplaced if it was sent in response to a request from a thread other than the currently active thread. Thread switches can also occur as the result of explicit application requests through a CVM system 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 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

	Input Set	*Sync Type*	*Modifications*
Barnes	10240 particles	barrier	g
FFT	64 x 64 x 64	barrier	-
Ocean	258 x 258 ocean	barrier, lock	g, r
SOR	2048 x 2048	barrier	-
Water-Sp	4096 molecules	barrier, lock	g, r
SWM750	750 x 750	barrier	-
Water-Nsq	512 molecules	barrier, lock	g, r, s

Table 1: Application specifics

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 in practice.

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 the results of running seven applications over four and eight processors with multi-threaded CVM. We first describe the environment and the applications, then proceed to discussions of the communication requirements, and the effects of multi-threading on DSM behavior, scalability, and the memory system. We finish with a case study of source modifications that attempt to improve the above interactions and reduce contention for resources.

4.1 Experimental Environment

We ran our experiments over CVM's lazy multi-writer protocol on a cluster of eight Alpha 2100 4/275 nodes. Each Alpha node has four 275 MHz Alpha processors and 256 Mbytes of memory. We used only a single processor per node in order to avoid contention at the network interface. The Alphas run Digital UNIX V4.0, and are connected via Digital's Gigaswitch/ATM communications hub. Each node currently has a 155 MBit/sec ATM interface.

CVM runs on UDP/IP over the ATM. 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 1100 µsecs in average. 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

Our application suite consists of seven applications: Barnes, FFT, Ocean, Water-Sp, and Water-Nsq from the Splash suite [10], SOR, a simple nearest-neighbor application, and SWM750 from the SPEC92 benchmark suite.

Table 1 lists specifics for the applications in our study. `Sync Type` indicates the synchronization operations used by the applications. The `Modifications` column shows three types of code modifications that we made, together with the reasons that the modifications were made:

g - *correctness* - Global program variables are accessible by local threads on each node. Hence, we disallowed the use of any global variables that are modified after the initialization phase. If a global variable was originally modified during execution, we then privatized that variable for each thread on a local node.

r - *performance* - As discussed in Section 2, reduction operations can become bottlenecks when multi-threading is added. We modified the applications to use per-node barriers and update remote data only once per node.

s - *performance* - Parallel shared memory programs often partition work by assigning different ranges of the same array to each thread. We enhance load-balancing among local threads by allowing all threads to access the entire array. Individual elements are protected from simultaneous modification through the use of local flags. This technique is not applicable to all the applications in our suite.

Barnes is a modified version of the gravitational N-body simulation from Splash-2. FFT is a 3-D Fast Fourier Transform that uses matrix transposition to reduce communication. Ocean (contiguous ocean from Splash-2) simulates large scale ocean movements based on eddy and boundary currents. `SOR` implements successive over-relaxation uses nearest neighbor communication. SWM750 performs a two dimensional stencil computation that applies finite-difference methods to solve shallow-water equations. This application was automatically parallelized by the SUIF compiler [4]. Finally, Water-Nsq and Water-Sp are molecular dynamics simulations from Splash-2. While the Water-Nsq uses $O(N^2)$ algorithms, the Water-Sp uses a more efficient algorithm by imposing a uniform 3-D grid of cells on the problem domain.

The single-threaded versions of these applications achieve eight-processor speedups (versus uniprocessor speedup without CVM calls) of 2.56, 2.29, 8.42, 1.86, 2.77 and 2.22 for Barnes, FFT, SOR, SWM750, Water-Nsq and Water-Sp and 0.33. These relatively low speedups are primarily due to the combination of very fast processors and an inefficient OS communication implementation.

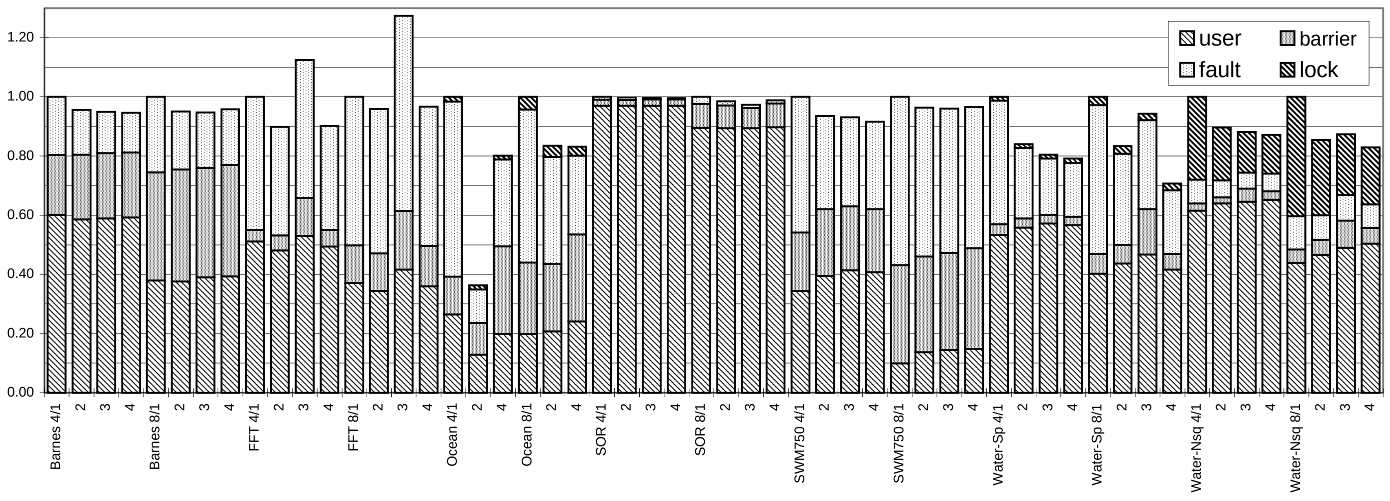

Figure 1: Normalized Execution Time on 4 and 8 Processors

4.3 Overall Results

Figure 1 shows the performance of the applications for four and eight processors, and from one to four threads, normalized to single-threaded execution times. Note that there is no three-thread case for Ocean because this application requires the number for threads to be a power of two. Ocean, Water-Nsq and Water-Sp achieved large multi-thread speedups. On eight processors, they sped up more than 17% with two threads, and 20% with four threads. Barnes, FFT, and SWM750 also improve by approximately 5% on eight processors and more on four. SOR, however, sped up only 2% on eight processors.

Figure 1 also breaks this speedup into contributions from user time (which includes all local consistency time), time spent waiting at barriers, non-overlapped time spent waiting on faults, and non-overlapped time spent waiting for locks. In general, multi-threading reduces fault and lock time by allowing other threads to run when the current thread blocks on a remote request. However, we have found that multi-threading tends to increase load imbalance. This increase is primarily caused by the fact that there is a great deal of variation among processors in how successful they are at hiding remote latency. User times vary because we are not currently accounting for the cost of handling remote requests that arrive while application code is executing.

Barnes achieves about 5% multi-thread speedup in all tests on both four- and eight-processor cases, primarily from reductions in fault time. This version of Barnes differs from the Splash version in that only barrier synchronization is used. Shared updates that were guarded by locks are now either serialized or partitioned among the processors.

Neither FFT nor SWM750 sped up significantly on eight processors, although on four processors they sped up by approximately 9%. The spike that appears at three threads in FFT is caused by poor alignment of data on pages and consequent load imbalance. More discussion of these effects can be found later in section 4.3.2. The increase in user time in SWM750 results from additional run-time routines that implement the SUIF fork-join model.

Another barrier-only application, SOR, improves by only 2% on eight processors. Since SOR's speedup is near-linear even in the single-threaded case, we did not expect it to improve significantly. We've included it primarily to show that our multi-threaded implementation imposes little additional overhead, even when there is very little remote latency to hide.

Water-Sp and Water-Nsq sped up by more than 10% on both four and eight processors. On eight processors and four threads, Water-Sp sped up by 41%, and Water-Nsq by 24%. Both applications made gains in both fault and lock time. However, most of Water-Sp's speedup is from faults, while most of Water-Nsq's is from locks.

Ocean performs poorly on CVM due to the large number of faults. On our single-thread case, it slowed down by approximately a factor of three. Although the multi-thread executions have a similar number of faults, much of the fault latency is hidden by thread switching, and we get a large improvement in performance. Ocean was included primarily to show the effect of multi-threading on applications that are anything but well-tuned for our environment.

In general, two threads usually improve performance, but additional threads increase barrier imbalance and interact poorly with the underlying DSM. Our system can actually increase load imbalance if the system is more effective at overlapping computation with communication at some nodes than others.

4.3.1 Effect on Communication

Table 2 shows the effect of multi-threading on the communication performance of CVM. The *Barrier*, *Lock*, and *Diff*

		Total Delay (msec)			*Total Messages*				*BW.*
	T	*Barrier*	*Lock*	*Diff*	*Barrier*	*Lock*	*Diff*	*Total*	*Kbytes*
Barnes	1	33379	0	23221	112	0	17828	17940	56522
	2	34510	0	17878	112	0	17866	17978	56593
	3	33798	0	17015	112	0	18065	18177	56683
	4	34286	0	17152	112	0	18278	18390	57679
FFT	1	10118	0	39672	203	0	6958	7161	64669
	2	10150	0	38533	203	0	7013	7216	64683
	3	15622	0	52177	203	0	11609	11812	110927
	4	10740	0	37260	203	0	7024	7227	64704
Ocean	1	29714	5338	63617	6314	2883	46950	56147	243205
	2	28077	4669	44567	6286	2882	47747	56915	224946
	4	36188	3836	32700	6364	2910	56192	65466	184637
SOR	1	13573	0	3868	1211	0	1162	2373	36498
	2	12637	0	2400	1211	0	1162	2373	36498
	3	11305	0	1850	1211	0	1162	2373	36498
	4	13382	0	1767	1211	0	1162	2373	36498
SWM 750	1	24	0	42	1533	0	8815	10348	44556
	2	24	0	37	1533	0	8808	10341	44523
	3	24	0	36	1533	0	8814	10347	44523
	4	25	0	35	1533	0	8807	10340	44523
Water-Sp	1	10951	4569	82009	220	723	68444	69387	141756
	2	10373	4247	50162	217	719	68445	69381	141820
	3	24902	3501	49108	236	723	79188	80147	158608
	4	8682	3791	35004	216	723	57691	58630	124632
Water-Nsq	1	3628	26473	7361	266	22035	3297	25598	19582
	2	3801	16611	5280	266	22034	3892	26192	19791
	3	6284	13502	5691	266	22001	4634	26901	20051
	4	3141	12692	5238	271	22034	4532	26837	20086

Table 2 : Communication Performance

columns show the remote latency that could not be overlapped with local computation. We expected lock and diff wait times to reduce because of the direct effect from multi-threading. On the other hand, total barrier wait time is more difficult to predict because our scheduler does not control loads that may be changed dynamically. Furthermore, because our applications distribute work load by dividing problem size with total number of nodes (total number of threads in this case), we expected barrier wait times to increase when using three threads per node. This matches the results from FFT, Water-Sp and Water-Nsq.

The *Messages* columns reflect the total of each type of message, and the *BW* column shows the total communication bandwidth requirements. *Diff* messages are used to satisfy remote data requests. The *Lock* column shows that there is essentially no change in the number of lock messages as the degree of multi-threading increases. This implies that we are able to successfully aggregate all local synchronization accesses to a given lock into a single remote access. This conclusion is supported by the *Block Same Lock* column in Table 3, which shows that we never had multiple threads block on the same lock.

The slight increases in diff messages for Barnes, SOR, SWM750, and Water-Sp reflect the small increases in their bandwidth. FFT's diff messages rise slowly from one to two to four threads, with a spike at the three-thread case. Finally, the number of diff messages increases dramatically for Ocean and Water-Nsq, for example, about 20% and 40% with four threads for Ocean and Water-Nsq respectively. Generally, increase of diff messages can result from bad page alignment and finer grain created by per-node multithreading. We will discuss this further in Section 4.3.2, and use Water-Nsq as a case study in Section 4.5.

4.3.2 Effect on DSM Consistency Actions

Table 3 shows details of the effect of multi-threading on the low-level behavior of the DSM. *Thread switch* is the total number of useful thread switches. The four-thread number varies from only 6394 in SOR, to 149493 in Water-Sp.

The rest of the table gives details of remote page and lock request overlapping. *Remote Faults* and *Remote Locks* list the total number of faults and lock acquires that require network communication. *Outstanding Faults* and *Outstanding 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 a remote request is initiated. Directly measuring the overlap of communication and computation is difficult to measure because we have no way of determining exactly when replies arrive in our system. However, these quantities do measure the frequency of multiple requests being outstanding at the same time. For example, if thread T_1 blocks on a remote page fault and the system switches to T_2, these statistics are not incremented. However, if thread T_2 then blocks on either a remote lock or remote fault, the *Outstanding Faults* will be incremented to reflect the fact that T_1 has an outstanding fault. In all cases except the three-thread FFT, outstanding faults and locks uniformly increase as the multi-threading level increases.

Block Same Page and *Block Same Lock* are the number of times multiple threads blocked on the same page or lock. As such, they give one measure of local contention for shared

		Thread	*Remote*		*Outstanding*		*Block Same*		*Diffs*	
	T	*Switches*	*Faults*	*Locks*	*Faults*	*Locks*	*Page*	*Lock*	*Created*	*Used*
Barnes	1	0	7727	0	0	0	0	0	4911	22991
	2	28446	7711	0	7472	0	1052	0	4960	23040
	3	37542	7707	0	14045	0	1964	0	5009	23091
	4	46034	7772	0	18913	0	3171	0	5058	23488
FFT	1	0	7360	0	0	0	0	0	6016	7070
	2	17206	7268	0	1527	0	5992	0	6016	7070
	3	28901	12284	0	10832	0	8264	0	6731	12140
	4	33312	7257	0	4532	0	17976	0	6016	7070
Ocean	1	0	47290	643	0	0	0	0	27852	59835
	2	87382	45196	632	37267	0	3773	0	23632	60914
	4	137898	47056	648	184565	0	14032	0	32759	61252
SOR	1	0	4783	0	0	0	0	0	1162	1162
	2	6394	4286	0	6483	0	22	0	1162	1162
	3	7848	4284	0	10089	0	44	0	1162	1162
	4	9272	4284	0	14382	0	66	0	1162	1162
SWM 750	1	0	11282	0	0	0	0	0	6050	16365
	2	19484	10350	0	9400	0	3152	0	6051	16372
	3	28635	10305	0	12752	0	6345	0	6052	16379
	4	36787	10269	0	16300	0	9497	0	6051	16372
Water-Sp	1	0	72758	96	0	0	0	0	25266	69342
	2	99262	72466	94	79311	3816	6346	0	25267	69349
	3	149493	83136	95	176222	7191	6329	0	27059	80101
	4	97019	61434	96	218615	9900	3365	0	28848	58576
Water-Nsq	1	0	3320	10261	0	0	0	0	1801	1801
	2	19716	3692	10260	2200	12645	464	0	2570	2570
	3	29027	3949	10252	3635	25219	1729	0	3449	3449
	4	32734	3713	10260	4786	37812	2530	0	3271	3271

Table 3: DSM Actions

resources. They also give a measure of how ill-suited the application is for transparent multi-threading. SWM750, for example, generated approximately 11,000 remote faults, and approximately 3100 *block same page*s for each additional thread. The obvious implication is that all threads are contending for the same resource. However, a *block same page* does not necessarily mean that no overlap was accomplished, as this measure gives no notion of how much computation was performed by the second thread before they also blocked.

Nonetheless, large numbers indicate that naively increasing the level of multi-threading is not likely to improve performance. As discussed in Section 2, there are basically two ways of dealing with local resource contention: source modification and using more threads. Using more threads usually improves the probability that at least one thread always has work to perform. However, all threads in some lock-based applications will block on the same lock, no matter how many threads are added. In this type of application, only source modification will allow multi-threading to be useful.

As SOR is a nearest-neighbor computation, there are never any pages that are accessed by more than one local thread *and* a thread on a remote node. Hence, after initialization, local threads never block on the same remote request. A corollary of this is that fault time is nearly negligible even in the single-threaded case, so there is very little multi-threaded speedup. This is also true for Barnes and SWM750, which use barriers but have more remote fault latency to hide by multithreading. Both applications were able to get 5% speedup, compared to only 2% in SOR.

One of the performance problems that multi-threading can introduce in a multi-writer system such as CVM is an increased number of diffs. Recall that diffs are used to summarize modifications to a given page. Multi-threading may break the modification of a given page into modifications by different threads. If the threads either are on different nodes, or modify their parts of the data at different times, per-thread diffs may be created instead of a single combined diff. The *Diffs Created* and *Diffs Used* columns of Table 3 show that this problem is negligible for FFT and SOR. It is only a minor problem for Barnes and Water-Sp, as only approximately 10% more diffs are created in the four-thread case. However, Ocean with four threads generates 45% more diff creations. Also, Water-Nsq creates 46% more diffs in the two-thread case, and 86% more in the four-thread case.

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, as only the final result needs to be seen at other nodes. However, both the number of diff requests and total amount of communicated data go down in Ocean and Water, so we infer that the bulk of the diffs created by subdividing single-thread diffs are non-overlapping.

4.3.3 Effect on Memory System

This section describes the effect of multi-threading on cache and TLB performance. We ran all of our experiments on both the Alphas and an IBM SP-2, but omitted the SP-2 numbers because of space limitations. However, we only have cache and TLB miss information for the SP-2, so we include them in the paper.

Figure 2 shows 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). 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 8Kbyte 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 cache. Hence the cache effects in Figure 2 are probably more pronounced than on the Alpha cluster.

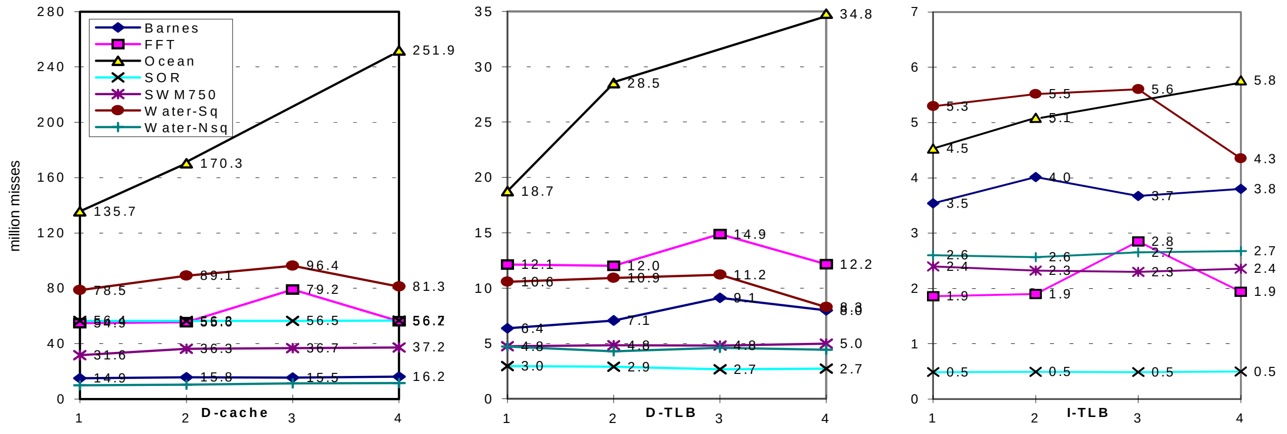

Figure 2: Effect on Memory System When Increasing Number of Threads

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 Water-Sp, which has fewer TLB misses at four threads than at one. Ocean's poor locality is caused by the large number of thread switches. Water-Sp's good locality is probably due to the decreased message traffic and consequent decrease in operating system calls.

4.4 Scalability

Table 4 shows statistics on runs for two and four threads relative to the single-threaded cases, for up to sixteen processors. Barnes will not run with our default input size on sixteen processors. As we have only eight machines, we had to run multiple copies of CVM on each node. Therefore, raw performance numbers would not be meaningful.

	P	*T*	*Total Msgs*	*BW Kbytes*	*Remote Fault*	*Diffs Created*		*P*	*T*	*Total Msgs*	*BW Kbytes*	*Remote Fault*	*Diffs Created*
FFT	4	2	0%	0%	-1%	0%	SWM750	4	2	0%	0%	0%	0%
		4	0%	0%	-1%	0%			4	0%	0%	0%	0%
	8	2	1%	0%	-1%	0%		8	2	0%	0%	0%	0%
		4	1%	0%	-1%	0%			4	1%	0%	-1%	0%
	16	2	0%	0%	-1%	0%		16	2	0%	0%	-1%	0%
		4	0%	0%	-1%	0%			4	-1%	0%	-1%	0%
Ocean	4	2	-48%	-65%	-57%	-62%	Water-Sp	4	2	0%	0%	0%	0%
		4	-18%	-45%	-36%	-33%			4	0%	0%	-1%	0%
	8	2	-1%	-11%	-7%	-13%		8	2	0%	-2%	-1%	0%
		4	16%	-24%	0%	18%			4	-20%	-18%	-20%	4%
	16	2	2%	-21%	-3%	18%		16	2	0%	-2%	-1%	0%
		4	28%	-3%	19%	74%			4	-18%	-18%	-19%	0%
SOR	4	2	0%	0%	-5%	0%	Water-Nsq	4	2	1%	0%	1%	18%
		4	0%	0%	-5%	0%			4	3%	1%	6%	43%
	8	2	0%	0%	-10%	0%		8	2	3%	2%	14%	58%
		4	0%	0%	-10%	0%			4	6%	3%	19%	101%
	16	2	0%	0%	-18%	0%		16	2	5%	4%	20%	95%
		4	-1%	0%	-18%	0%			4	11%	7%	29%	159%

Table 4: Scalability

Instead, we show total messages, bandwidth consumed, remote misses, and diffs created for each number of threads and processors. While this information does not give us complete information on scalability, it does allow us to gauge the effect of multiple threads and increasing numbers of processors on the underlying DSM protocol. As the performance of this protocol is crucial to overall performance, we expect that raw performance would be consistent with the numbers that we present.

Numbers in the table reflect multi-threaded cases relative to the single-threaded cases, so positive numbers mean that the quantities are increasing. All four quantities have a negative effect on performance. Therefore, quantities that increase with number of threads *faster* on sixteen processors than on four are quantities that are not scaling well. This is not the case for any of the applications except Ocean, which has a slowdown on our multi-processor system. Increasing the number of threads actually seems to benefit Water-Sp more at larger numbers of processors than at smaller.

We therefore conclude that the interaction of multi-threading with the underlying DSM is not affected by the number of processors in the system. Hence, the scalability of the multi-threaded system should be no worse than the scalability of the uni-threaded system.

4.5 Case Study: Water-Nsq

Table 5 shows the effects of two source-level modifications that we made to the Water-Nsq application. Water-Nsq's primary data structure is a large array that describes force and position information for each molecule. Responsibility for updating molecules is partitioned among threads in the system, but all threads usually read all molecules at some point during each iteration.

No Opts refers to the version that differs from the default Splash program only in that global variables were promoted to shared variables in order to allow multiple threads to co-exist on a single node. Multi-threading uniformly hurts the performance of this version of Water-Nsq. The problem is similar to the example in Section 2. Naively multi-threading the work at each node results in multiple local threads acquiring the same locks and modifying the same shared data at different times, leading to extra diff creations and messages. The last two columns show that diff creations and uses rise with the number of threads. Although messages are not shown in Table 5, the table shows that the number of remote fault and lock requests uniformly rises with multi-threading level. Additionally, the *Block Same Page* and *Block Same Lock* columns show that the majority of lock acquires, and roughly half of the page faults, block more than a single thread at a time. This leads to reduced overlapping of communication and computation.

The second version, *Local Barrier,* includes optimizations in four routines that aggregate local updates to shared data structures. A CVM-provided local barrier stalls threads until all have arrived. All threads then cooperate in applying local updates to the global array. Each thread starts at a different portion of the shared array, wrapping around to sections being handled by other threads if they finish first. This is actually a crude form of load-balancing among local threads, and helps to increase overlapping of remote requests. Table 5 shows that this optimization dramatically reduces the number of thread switches. It also increases the number of overlapped page faults and lock acquires, reduces the number of *Block Same*

		Thread		*Remote*		*Outstanding*		*Block Same*		*Diffs*	
	T	*Spdup*	*Switch*	*Faults*	*Locks*	*Faults*	*Locks*	*Page*	*Lock*	*Created*	*Used*
No Opts	1	0.0%	0	3454	10320	0	0	0	0	1874	8590
	2	2.3%	40312	3704	10366	840	3389	1576	9227	1979	9199
	3	-5.1%	66016	4006	10391	1457	6793	3248	18383	2345	10770
	4	0.1%	90305	4052	10428	1890	10691	5037	27812	2270	10664
w/ Local Barrier	1	0.0%	0	3352	10261	0	0	0	0	1801	8075
	2	17.6%	20594	3706	10260	1452	12645	1178	0	2532	11114
	3	14.3%	30407	4031	10276	2796	25265	2991	0	3583	15399
	4	20.1%	34645	3753	10254	3565	37794	4249	0	3304	14261
w/ Both Opts	1	0.0%	0	3320	10261	0	0	0	0	1801	8052
	2	17.8%	19716	3692	10260	2200	12645	464	0	2570	11249
	3	15.2%	29027	3949	10252	3635	25219	1729	0	3449	14820
	4	24.6%	32734	3713	10260	4786	37812	2530	0	3271	14085

Table 5: Water-Nsq Optimizations

*Page*s, and eliminates all *Block Same Lock*s. These gains are somewhat mitigated by the dramatically increased number of diffs created and used.

We attempted to take this modification further by making the threads conscious of page alignment when deciding which molecules *owned* by other threads to help compute. While this eliminated most *Block Same Page*s, it also increased diff creations to the point where they overwhelmed the gains and reduced multithread speedup.

The final set of rows in Table 5 shows the result of combining the local barrier optimization with a reordering of molecule accesses performed in another part of the application. Even though threads in this portion of the program update disjoint sets of molecules, they essentially read all of them. This optimization orders the reads so that they start at opposing ends of the molecule array. The change delays the occurrence of overlapping reads to the same page by multiple threads as much as possible, at least for two threads. The result is increased remote request overlap and decrease *Block Same Page*s, without significantly affecting diff creations or the total number of remote requests. This is the version discussed in the rest of the paper.

5. Conclusions and Future Work

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%, and all gained some benefit from the multi-threading. We identify the following as limiting factors in multi-thread speedup:

1. *Local contention for resources* - Transparently adding per-node multi-threading to explicitly-parallel applications often results in multiple threads blocking on the same resource. This can prevent or limit the overlapping of remote requests and local computation.
2. *Interactions with the underlying DSM* - Splitting accesses to a single page among multiple local threads can lead to the threads accessing the data at different times. This can result in multiple local diffs or remote page faults where only a single one sufficed for the single-thread case. Unless the multi-threading is uniformly good or uniformly bad at hiding remote latencies, the tendency is to increase load imbalance and barrier wait times.
3. *Interactions with the memory system* - Multi-threading increases the pressure on both caches and TLBs. Without cache and TLB-conscious thread scheduling, the memory system may be the ultimate bottleneck in multi-threading performance, especially with software DSMs.
4. *Thread switch cost* - Efficient thread switching is crucial to getting good coverage of remote latency.

One obvious approach to solving the majority of the above problems is to combine a lightweight, fine-grained threading package with adaptive load-balancing [11]. Lightweight thread packages [6] are 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 threads [12]. The challenge is to build a lightweight threading system without changing the programming model, i.e. without constraining the threads to be *run-to-completion*. We are continuing our research in this direction.

References

1. Keleher, P. “The Relative Importance of Concurrent Writers and Weak Consistency Models” in *Proceedings of the 16th International Conference on Distributed Computing Systems*. 1996.
2. Agarwal, e.a. “The MIT Alewife Machine: Architecture and Performance” in *Proceedings of the 22th International Conference on Computer Architecture*. May 1995.
3. Mowry, T. and A. Gupta. “Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors” in *Journal of Parallel and Distributed Computing*. June 1991.
4. Wilson, R.P., *et al.*, “SUIF: An Infrastructure for research on parallelizing and optimizing compilers”, *ACM SIGPLAN Notices*, December 1994. **29**(12): p. 31-37.
5. Lim, B.-H. and R. Bianchini. “Limits on the Performance Benefits of Multithreading and Prefetching” in *Proceedings of the International Conference on the Measurement and Modeling of Computer Systems*. 1996.
6. Freeh, V.W., 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*. November 1994. Monterey, CA: USENIX Assoc.
7. Weimin Yu Cristiana Amza, A.L.C., Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony and W. Zwaenepoel. “TreadMarks: Shared Memory Computing on Networks of Workstations” in *IEEE Computer*. February 1996.
8. Gharachorloo, K., *et al.* “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*. May 1990.
9. Keleher, P., 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.
10. Woo, S.C., *et al.* “The SPLASH-2 Programs: Characterization and Methodological Considerations” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. June 1995.
11. Itzkovitz, A., A. Schuster, and L. Wolfovich, Thread Migration and its Applications in Distributed Shared Memory Systems, . 1997, Technion IIT.
12. Philbin, J., et al. “Thread Scheduling for Cache Locality” in *Proceedings of the 7th International Conference on Architectural Supports for Programming Languages and Operating Systems*. 1996.