Update Protocols and Cluster-based Shared Memory

Peter J. Keleher
keleher@cs.umd.edu
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Software DSMs have been a research topic for over a decade. While good performance has been achieved in some cases, consistent performance has continued to elude researchers. This paper investigates the performance of DSM protocols running highly regular scientific applications. Such applications should be ideal targets for DSM research because past behavior gives complete, or nearly complete, information about future behavior. We show that a modified home-based protocol can significantly outperform more general protocols in this application domain because of reduced protocol complexity.

Nonetheless, such protocols still do not perform as well as expected. We show that the one of the major factors limiting performance is interaction with the operating system on page faults and page protection changes. We further optimize our protocol by completely eliminating such memory manipulation calls from the steady-state execution. Our resulting protocol improves average application performance by a further 34%, on top of the 19% improvement gained by our initial modification of the home-based protocol.

Keywords: DSM, Shared Memory, Update Protocols, Scientific Computing.

1. Introduction

Iterative scientific applications would seem to be ideal applications from the perspective of a software distributed shared memory (DSM) system. While aspects of such programs differ, many have highly regular sharing behaviors. The set of shared data accessed by individual threads is often invariant from one iteration to the next. This regular behavior can be used by DSMs to predict future accesses, and to move data *in advance* of subsequent accesses [4, 11, 12]. Such update protocols allow much of the latency of remote data fetches to be hidden. Given reasonably efficient communication, DSMs should be able to achieve good speedups on such applications.

The output of parallelizing compilers, such as SUIF [13], is a good source for this type of application. While parallelizing compilers can analyze many sequential programs well enough to generate message-passing versions, this process is by no means easy or common. The process of generating parallel applications that run in shared memory environments requires far less analysis. Further, the set of applications that can currently be analyzed well enough to turn into a shared memory application is much larger than for message-passing applications.

By combining parallelizing technology with sophisticated runtime systems [3, 4, 12], we can create a programming environment that is flexible and easy to use. Scientists are not required to write message-passing programs or use data-parallel languages such as HPF. Instead, they can write sequential programs, rewriting a few computation-intensive procedures, and adding parallelism directives where necessary. This combination has the advantage of producing programs that can run on large-scale parallel machines as well as the more pervasive multiprocessor workstations. This portability is important for scientists and engineers who want to develop applications that run well on their multiprocessor workstations, but who desire the ability to scale their applications up for larger parallel machines as needed. The combination of ease of use and scalability of the resulting software is a key appeal of shared-memory compilers.

1.1 Contributions

This paper presents the design and performance of several new protocols to handle this and similar types of applications. We first look at a relatively straightforward update protocol based on multi-writer lazy release consistency (LRC) [6].

We then show that modified *home-based* [15] protocols can perform even better than LRC protocols for applications of this type. Whereas "home-less" LRC protocols can perform poorly for applications that modify (and communicate) large amounts of data, home-based protocols maintain relatively little state, and such state lives has short lifetimes. The main drawbacks of home-based protocols are related to problems adapting to dynamic sharing patterns, precisely the sort of pattern that the applications we are investigating here do not have.

The rest of the paper is organized as follows. Section 2 discusses the background of LRC and home-based protocols and describes the specific protocols that will be investigated in this work. Section 3 presents the performance of these protocols and attempts to relate their differences to how the protocols interact with specific sharing patterns. Sections 4 and 5 describe and analyze the performance of extensions to our home-based protocols that impose less of a load on the underlying operating system. Finally, Section 6 concludes.

2. Background and Protocol Descriptions

Our home-based protocols are based on those discussed by Zhou [15]. Home-based protocols are, in turn, based on the multi-writer lazy release consistent (LRC) protocols used by DSMs such as TreadMarks [1] and CVM [5].

2.1 Distributed Shared Memory (DSM) Systems

CVM is a software distributed shared memory (DSM) system. DSM allows processes to assume a globally shared virtual memory even though they execute on nodes that do not physically share memory. Figure 1 illustrates a DSM system consisting of *n* networked workstations, each with its own memory, connected by a net-

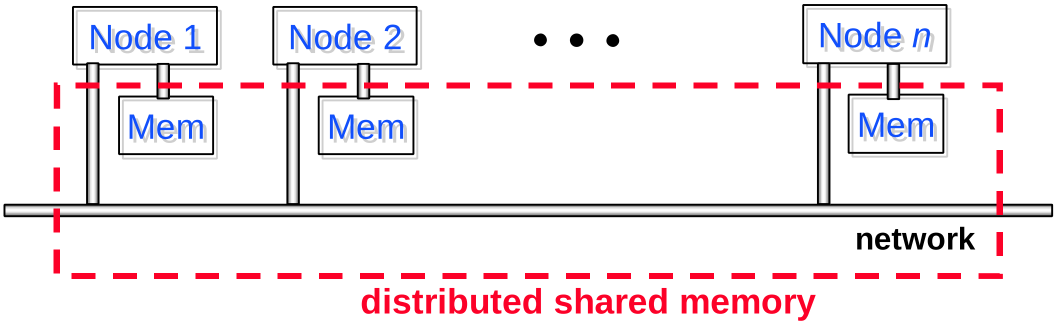

Figure 1: Distributed Shared Memory: The underlying DSM system uses local memories as caches for remote pages. The net result is that the local memories, in combination with the network, appear as an abstraction of shared memory.

work. The DSM software provides the abstraction of a globally shared memory, in which each processor can access any data item, without the programmer having to worry about where the data is, or how to obtain its value. In contrast, in the native programming model on networks of workstations, message passing, the programmer must decide when a processor needs to communicate, with whom to communicate, and what data to send.

For programs with complex data structures and sophisticated parallelization strategies, this can become a daunting task. On a DSM system, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values. In addition to ease of programming, DSM provides the same programming environment as that on (hardware) shared-memory multiprocessors, allowing programs written for a DSM to be ported easily to a shared-memory multiprocessor. Porting a program from a hardware shared-memory multiprocessor to a DSM system may require more modifications to the program, because the much higher latencies in a DSM system put an even greater value on locality of memory access.

The programming interfaces to DSM systems may differ in a variety of respects. We focus here on memory structure and memory consistency model. An unstructured memory appears as a linear array of bytes, whereas in a structured memory processes access memory in terms of objects or tuples. The memory model refers to how updates to shared memory are made visible to the processes in the system. The most intuitive model of shared memory is that a read should always return the last value written. Unfortunately, the notion of “the last value written” is not well defined in a distributed system. A more precise notion is sequential consistency [8], whereby the memory appears to all processes as if they were executing on a single multiprogrammed processor. With sequential consistency, the notion of “the last value written” is precisely defined. The simplicity of this model may, however, exact a high price in terms of performance, and therefore much research has been done into relaxed memory models. A relaxed memory model does not necessarily always return to a read the last value written.

2.2 Multi-Writer LRC Protocols

2.2.1 Lmw-i

LRC protocols allow the shared memory system to delay performing shared updates until specific synchronization events occur. For the applications in this study, this means the next barrier. The advantage of LRC in this context is that sharing (either true or false) does not cause invalidations between barriers.

This is useful in two cases. First, consider an example with true sharing. Shared data item *x* is initially valid on both nodes *i* and *j*. If process p_i writes to data *x* during the same barrier epoch (between the same two barriers) in which p_j reads *x*, the value returned by the read does not depend on the relative ordering of the read and the write. The value that is returned by the read is always the last value written prior to the previous barrier. By contrast, the canonical sequentially consistent [8] memory model would require node *j*'s copy of *x* to be invalidated before p_i is allowed to modify it. The result is that if the write occurs before the read in "wall-clock" time, the read will return the new value. However, if the read occurs before the write, the previous value is returned.

Consider where this is useful. If the programmer intended p_j's read to return the previous value of *x*, then an *anti-dependence* exists between the two accesses. The compiler would need to add an extra round of synchronization to enforce the anti-dependence for a sequentially consistent system. For an LRC system, on the other hand, the notice of the write is only propagated to p_j at the next barrier. The anti-dependence is therefore protected without recourse to extra synchronization.

The second case where LRC is useful for our applications is in hiding the effects of false sharing. Assume two data items *x* and *y* reside on the same shared page. Sequentially consistent systems require processes to gain exclusive access to shared pages before modifying any items that reside on the pages. Therefore, modification of *x* and *y* by distinct processes during the same barrier epoch requires the processes to communicate in order to arbitrate access to the page. By contrast, multi-writer LRC protocols allow the modifications to proceed in parallel, without communication. The separate modifications are merged at the next synchronization.

Lmw-i implements an invalidate-based multi-writer LRC protocol. Multi-writer protocols allow multiple processes to simultaneously modify the same page without network communication. Our programs are presumed to be free of data races, so such concurrent modifications are constrained to be to disjoint sections of the pages. At the end of each barrier epoch, the modifications to each shared page are captured in the form of *diffs* (Figure 2). A diff is a run-length encoding of the changes made to a single virtual memory page. Diffs are created by performing a page-length comparison between the current contents of the page and copy 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.

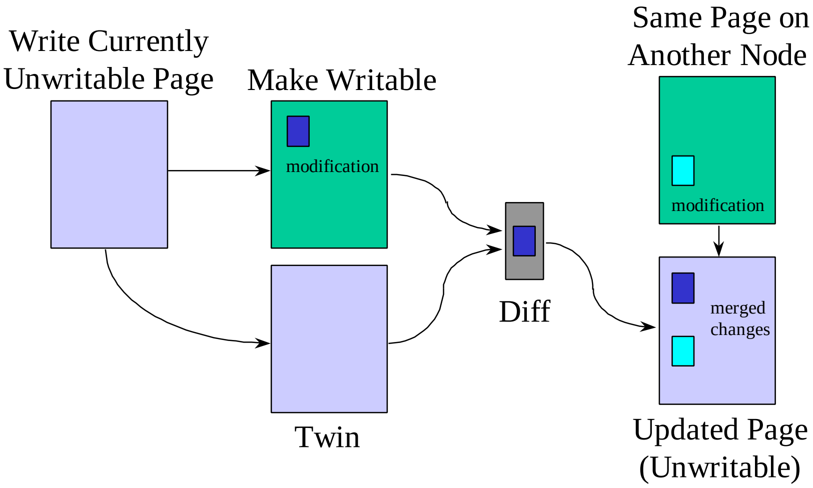

Figure 2: Diff Creation: Updates are communicated in the form of *diffs*. Pages are initially write-protected. At the first write access, the system makes a copy, or *twin*, of the unmodified page. After the modification is complete, a diff is created by comparing the twin with the page's current contents. The diff consists of all modified data on the page. Later, other page replicas are brought up to date by the *applying* the diff.

Since Lmw-i is an invalidate protocol, these diffs are not immediately sent to other processes. Instead, structures called write notices are distributed to other processes via existing synchronization (barrier) messages. Each write notice informs the recipient that a shared page has been modified, and the recipient invalidates any local copy of the page. The write notice also names the diff that needs to be applied in order to bring the local copy of the page up to date. Accesses to invalid pages cause page faults, during which the diffs named by write notices are retrieved and applied to the faulting page. Once the diffs have been applied, the page protections are re-validated and the application continues.

2.2.2 Lmw-u

Lmw-u is an extension of lmw-i that uses updates to communicate data whenever possible (see Figure 3). A more complete description can be found in the literature [12], but we provide a short description here.

Accesses to shared pages are tracked by using per-page copysets, which are bitmaps that specify which processors cache a given page. This information can be used to selectively employ a hybrid invalidate/update coherence protocol. Coherence for pages that are consistently communicated between the same set of processors can be updated, rather than invalidated, after writes. Such updates eliminate page faults. Coherence for the remaining pages is maintained by using the above invalidate protocol.

On the first iteration of each time-step loop, copysets for all pages are empty, and page faults can occur. By the second iteration, however, copyset information accurately reflects stable sharing patterns by indicating the processors that need each page. Page faults can be then be eliminated by sending any local updates to all

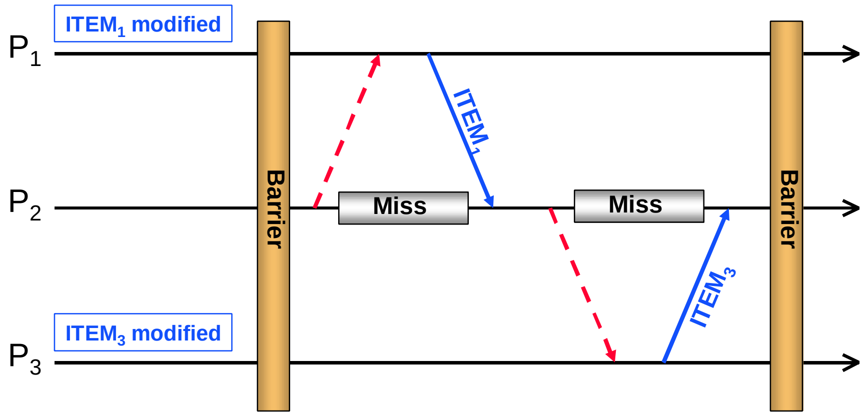

Figure 3: Update protocols – If the modified pages must be fetched on demand, the entire latency of the remote requests is seen by the requesters. However, if *item*$_1$ and *item*$_3$ are pushed by P_1 and P_3 prior to the first barrier, no miss latency is incurred.

processors on the copyset for each page. Since updates are sent before the data is needed, subsequent remote page faults are avoided.

SUIF was modified to automatically insert calls to DSM routines to mark pages to be updated at barriers. For a given page, local modifications are then flushed to all other processors in the page's local copyset at each barrier. A processor *p* is inserted into processor *q*'s copyset for a page if *p* requests a diff for the page, or if *q* sees a write notice for the page that was created by *q*.

Compiler analysis needed to use such a protocol is much simpler than communication analysis needed in HPF compilers. The identities of the sending/receiving processors do not need to be computed at compile time. Instead, the compiler only needs to locate data that will likely be communicated in a stable pattern, then insert calls to DSM routines to apply the update protocol for those pages at the appropriate time. More precise compiler analysis can be used to explicitly clear or set the copysets of data to be communicated. The compiler’s annotations do not need to be guaranteed correct, since the page annotations only affect efficiency, not program correctness.

As previously discussed, barrier flushes of updates (essentially a restricted update model) have both advantages and disadvantages. On the plus side, flushes ideally move data before it is needed, allowing computation and communication to be wholly overlapped. The result can be fewer page invalidations and page faults. A second advantage is that lost flush messages do not affect correctness, only performance. Flush messages can be unreliable, and therefore do not need to be acknowledged. A ``flush" therefore consists of only a single message, whereas a miss to shared data incurs at least one request and response message pair.

All consistency information in lazy-release-consistency systems is piggybacked on synchronization messages (barrier messages in the case of compiler-parallelized applications). By contrast, diff requests are inherently two-way, and so cost two messages. On the negative side, if sharing patterns are not stable, out-of-date

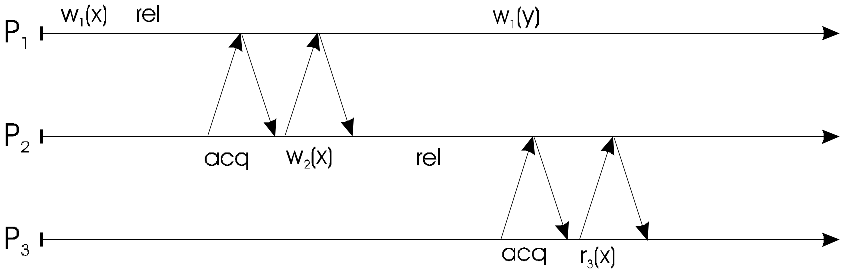

Figure 4: Diff Exchanges

copysets will cause data to be sent to processors that do not need it. Correctness is not affected, but the unneeded flushes cause unnecessary overhead.

2.3 Home-Based Protocols

Home-based protocols differ from "homeless" protocols, such as lmw-i and lmw-u, in that each page has a statically assigned *home*. Each update to a page is flushed to the home at the next synchronization. The result is that the “home” copy of each page is always up to date, at least with respect to the synchronization operations that have been performed at it.

The home-based protocol has two potential advantages: the “home” effect, and the short lifetimes of many data structures. As discussed above, homeless protocols summarize modifications in the form of diffs. Aside from the expense of creating and applying diffs, they can cause homeless protocols to have voracious appetites for memory because diffs have long lifetimes. The “home” effect refers to the fact that home-based protocols allow the owner of a page (i.e. the home) to dispense with creating diffs describing its own modifications. Diffs are created only to describe modifications made to a page by processes other than the page’s home node. Modifications made by the home node are merely noted locally. No network communication is required.

Second, home-based diffs have short lifetimes. The most inelegant aspect of homeless protocols is that the data structures that describe shared modifications can not be discarded until explicitly garbage-collected. For example, consider Figure 4. This example shows three processes, P_1 through P_3, that access migratory data *x*. First, P_1 modifies *x* and releases a synchronization variable. P_2 then acquires the synchronization variable, causing any local copy of the page that contains *x* to be invalidated. Lastly, P_2 accesses *x*.

With a homeless protocol, touching *x* causes a page fault that is satisfied by requesting the diff of $w_1(x)$ from P_1. However, the diff can not be discarded by P_1 even after it has been supplied to P_2, because P_1 can not know if or when some other process (P_3, for example) might subsequently request the diff as well. More generally, the diff can not be discarded until the system can guarantee that no process will request it in the future. The situation is complicated even more by the fact that if and when P_3 requests $diff_1$, the diff may be requested from P_2 rather than P_1. For performance reasons, then, P_2 can not discard the diff either. The result is that no diff, nor any of the write notices that name diffs, can be discarded until garbage-collection occurs.

By contrast, diffs have very short lifetimes under home-based protocols. Diffs are created at synchronization points, flushed to the home nodes, and immediately discarded. This is correct under home-based protocols because all page faults are serviced via complete page copies from the home node, rather than by applying diffs to pre-existing page replicas.

Consider Figure 4 again. Assuming that the manager of the affected page is P_3, both P_1 and P_2 will create and send diffs to P_3 prior to their releases. P_2 will also need to request a new copy of the page from P_3 before it can perform the local modification. The advantage is that both diffs can be immediately discarded. The disadvantage of this approach is that more messages are sent than with homeless protocols. The situation can be even worse. Consider the case where a fourth process, P_4, is the home node for the page. In this case, both P_1 and P_2 will send diffs to $P_{4.}$ Both P_2 and P_3 will then request copies of the page from P_4, a node that isn't even involved in the communication. This behavior means that home assignment must take expected sharing behavior into account in order to get efficient communication. Furthermore, dynamic sharing patterns would require additional mechanisms in order modify home assignments.

To summarize, home-based protocols have the following advantages:

1. Modifications made by the home node do not require diffs to be created. They do need to be tracked, however, and so still require local interrupts the first time each page is modified during each barrier epoch.
2. Validating a page always requires exactly one request-reply pair since validations are always accomplished by retrieving a complete new copy of the page.
3. There is very little persistent state. No garbage collection is needed.

There are also several obvious disadvantages:

1. The home node must be chosen wisely, and the application must not change sharing patterns drastically.
2. Communication of data between two non-home processes requires the data to be sent through the home node. Consider a piece of migratory data *x*. Each time *x* moves from one process to the next, it must be sent back to the home node in the form of a diff, and then paged in from the second node. By contrast, the data travels directly from one process to the next in a homeless protocol.
3. Potentially more diffs are created, as diffs are created promptly at the end of each interval rather than lazily [7], as with homeless protocols.

2.3.1 Bar-i and Bar-u

In order for the "home" effect to be useful, modifications of shared data should be mainly done by the data's home. This is somewhat analogous to the "owner computes" rule often used in parallelizing compilers. Unsurprisingly, therefore, the output of parallelizing compilers is a good candidate for a home-based protocol. Our

Apps	Input	Summary
barnes	16k bodies	n-body gravitational simulation (SPLASH)
expl	512 x 512	explicit hydrodynamics (Livermore 18)
fft	64x64x64	3-D fast Fourier transform (SPLASH)
jacobi	1024 x 1024	jacobi w/ convergence test
shallow	750 x 750	shallow water model (RICE)
sor	2k x 1k	red-black jacobi
swm	750 x 750	shallow water model (SPEC)
tomcat	512 x 512	vector mesh generation (SPEC)

Table 1: Application summary

bar-i protocol is a simplified home-based protocol that has several extensions to support iterative scientific codes.

First, by limiting the protocol to codes that only use barrier synchronization, we can prevent any diff or consistency state from living past the next barrier. Page coherence is maintained by using a per-page scalar version index, which is maintained by the page's home node. The index is incremented with any local modification of the page (but only once per barrier epoch), and for each applied diff from another node. Indexes for modified pages are distributed via the barrier process. The new page versions are used by other processes to decide which pages need to be invalidated.

Second, bar-i has been augmented to provide explicit support for reductions. Many of our applications were automatically parallelized by SUIF [13], and reductions are used in most of these codes. Accesses to shared pages are tracked by using per-page copysets, which are bitmaps that specify which processors cache a given page. This information can be used to selectively employ a hybrid invalidate/update coherence protocol. Using updates rather than invalidates allows page faults to be eliminated for pages that are consistently communicated between the same processes. Coherence for the remaining pages is maintained using an invalidate protocol in order to avoid excessive communication. On the first iteration of the time-step loop, the copysets of each page are empty and page faults occur. By the second iteration, however, copyset information indicates the processors that need each page, accurately reflecting stable sharing patterns.

Third, our barrier protocols assign page homes at runtime, rather than requiring the compiler or user to do so statically. The most obvious drawback of a scheme with statically assigned homes is that the initial assignment must be done well. Secondarily, the scheme will not adapt if an application undergoes a phase change; a point in the code where the set of pages written by a given process change.

Zhou [15] addressed the problem of assignments by requiring user annotations on each section of data, and observed that making such assignments is easy for the majority of cases. This is especially true for scientific applications, which tend to distribute computation among processes as large array slices. Even when this is the case, however, such annotations are an additional burden on the programmer.

Our protocols collect access behavior information during the first iteration of a program, and migrate pages before the second iteration begins. We migrate any pages that have not been written by their initial owner, but have been written by at least one other process. The migration decisions are distributed on release messages at the next barrier.

Similarly to lmw-u, then, page faults can be eliminated for bar-u by updating processors on the copyset for each page, sending the data before it is accessed.

Additional extensions are discussed in Section 4.

3. Experimental Results

3.1 Applications

The applications used in this study are summarized in Table 1. The shared segment size is the size of the shared portion of the address space, while “Sync. Gran.” is the average period between barrier synchronizations. Barnes is a version of the n-body simulation from SPLASH-2 [14] that has been modified to use less synchronization, and to perform some tasks (i.e. maketree) serially in order to reduce parallel overhead. Expl is a dense stencil kernel typical of those found in iterative PDE solvers. FFT is a three-dimensional implementation of the Fast Fourier Transform that uses matrix transposition to reduce communication. Jacobi is a stencil kernel combined with a convergence test that checks the residual value using a max reduction. SOR is a simple nearest-neighbor stencil. Shal and Swm are different versions of the shallow water simulation, differing primarily in synchronization granularity. Both swm and tomcat are programs from the SPEC benchmark suite and contain a mixture of stencils and reductions. We used the APR version of tomcat, in which the arrays have been transposed to improve data locality.

In all cases, speedups are calculated with reference to a single-process version of the same program with synchronization macros nulled out, allowing all traditional sequential optimizations to be performed. Additionally, we start timing only after the applications have reached a steady state (and after all page home assignments occur). There are two primary reasons for this. First, ignoring the initial iterations allows us to quickly approximate the behavior of the long runs typical of users of parallel systems, rather than developers. Second, the underlying OS can take a significant amount of time to settle when applications with large address spaces are started.

3.2 Experimental Environment

Our experimental environment consists of an 8-node IBM SP-2. The SP-2 has a high-performance switch (HPS) in which each bi-directional link is capable of a sustained bandwidth of approximately forty megabytes per sec-

	Diffs				Remote Misses				Messages				Data (kbytes)			
	li	lu	bi	bu	li	lu	bi	bu	li	lu	bi	bu	li	lu	bi	bu
barnes	3261	3261	2688	3274	4185	0	3789	0	16005	2269	4048	1968	28604	28918	33187	27106
expl	632	642	270	648	674	0	390	0	849	247	595	277	1912	1930	3423	1945
fft	2720	2464	140	2464	4640	0	4620	0	5627	2582	4767	1512	36545	41691	37339	32546
jacobi	179	198	77	220	251	0	210	0	412	293	404	293	1236	1294	2259	1543
shallow	5501	5929	2882	5929	6233	198	3420	0	8153	3637	5044	3439	1412	790	27890	783
sor	126	126	0	126	126	0	126	0	196	183	196	178	283	285	1024	264
swm	4408	4858	4873	7462	5159	0	2274	0	6062	2007	3709	2139	8798	9319	32218	19204
tomcat	898	899	413	911	1084	0	625	0	1343	547	992	541	3649	3600	5931	3890

Table 2 : Base Statistics

ond. Each processor is a 66MHz RS/6000 POWER2 and has 128 megabytes of memory. The applications were run on a version of CVM that uses UDP/IP over the high-performance switch.

Simple RPC's in our environment require 160 µsecs. Remove page faults require 939 µsecs. In the best case, AIX requires 128 µsecs to call user-level handlers for page faults, and mprotect system calls (which are used to change the protection level of individual pages) require a minimum of 12 µsecs. However, the costs of virtual memory primitives in the current system are location-dependent, occasionally increasing the cost of page protection changes by an order of magnitude. This phenomenon is explored in more detail in Section 4.

Although AIX’s default virtual memory page size is 4k bytes, we used 8k pages in CVM by the simple expedient of ensuring that all page protection changes use an 8k granularity. The larger page size generally increases performance by aggregating data into fewer messages [2].

3.3 Base Results

Figure 5 shows speedups of our application using four protocols, lmw-i, lmw-u, bar-i, and bar-u. Table 2 shows the number of diff creations, misses (remote faults) that cause network traffic, the number of data and synchronization requests sent (there are an equal number of replies), and the total amount of data communicated, in kilobytes. Protocols lmw-i, lmw-u, bar-i, and bar-u are abbreviated li, lu, bi, and bu, respectively.

Since all of our applications are repetitive scientific computations, the update versions of lmw and bar are almost uniformly faster than the invalidation versions. Both update protocols eliminate the majority of remote misses in most of the applications. The exceptions are barnes and swm, both of which perform much worse for the update version of lmw than for the invalidate version. In both cases, the poor performance is the result of frequent garbage collection. Lmw protocols discard updates only when a global garbage-collection algorithm is run. The update version tends to send all updates everywhere for these applications, resulting in most diffs being cached on most processors. The result is that the garbage collection algorithm is run frequently. The bar protocols do not suffer from this problem because all diffs are discarded after being flushed to the home.

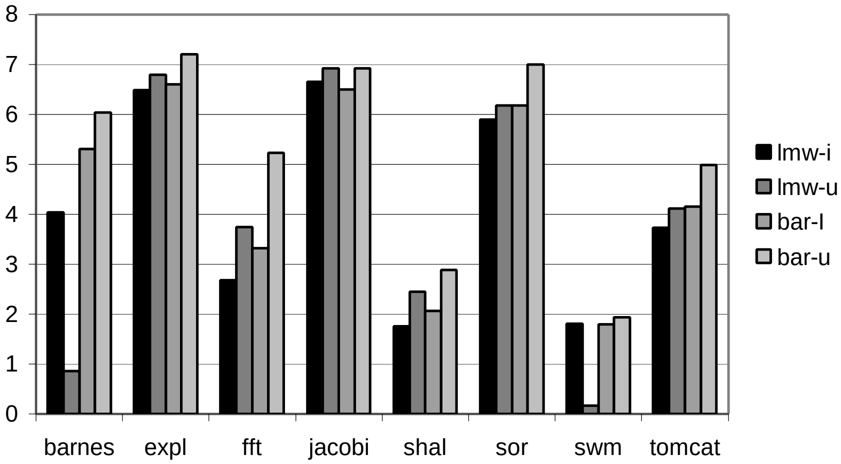

Figure 5: 8-Proc Speedups

Second, the home-based protocols outperform the homeless protocols for all but jacobi, which performs similarly for both update-based protocols. A number of factors contribute to this difference. First, the home effect allows the invalidate version of bar to create an average of 36% fewer diffs than the corresponding lmw protocol. This translates into 31% fewer remote misses, and a total of 49% fewer messages. This is partially offset by the fact that bar-i sends 74% *more* data. The reason is that lmw moves most data in the form of diffs, whereas bar-i satisfies all remote misses with complete new copies of pages. Diffs are usually much smaller than page size.

The complexion of these statistics changes when we look at the update protocols. bar-u uses diffs to push data before it is needed, just as with lmw-u. Since page faults no longer occur (with the sole exception of a small number for shallow running on lmw-u), the total amount of data moved is almost identical between the two update protocols, approximately equal to the amount moved by lmw-i. bar-u creates about 14% more diffs than lmw-u and sends about 12% fewer messages. Overall, bar-u averages approximately 19% more speedup than the better of the two lmw protocols.

Although the message count differential can certainly account for some of the performance difference between the update protocols, there are at least two other factors at work. The first is the sheer complexity of the homeless protocols. Deciding what consistency actions need to be performed consists of filtering locally known consistency actions by what is known of the node that is being synchronized with. Since lmw supports locks, flags, and other non-global synchronization types, as well as programs with dynamic sharing behavior, consistency information has long lifetimes, and can not be discarded without explicit garbage collection.

Furthermore, nodes running lmw often receive diffs from nodes other than the diffs' creators (see [1] for more details). The result is that in the general case, the producer of a diff often has limited information about

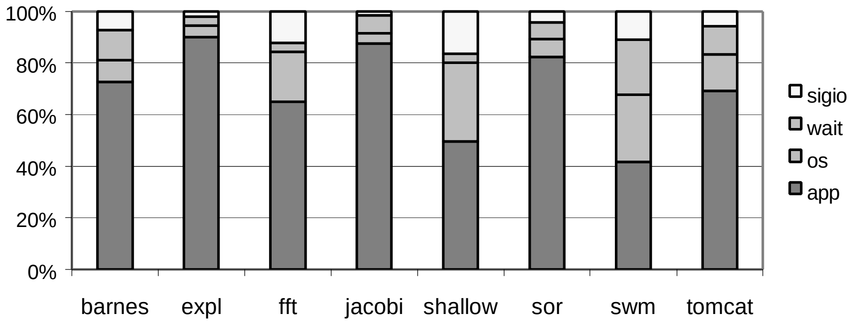

Figure 6: Time Breakdown for Bar-u

consumers of the diff. This has the effect of making copysets (and hence updates) less precise. As a consequence, lmw-u does not immediately validate pages when diffs that make it possible arrive by update. Instead, lmw merely stores updates to locally invalid pages and checks to see if all required diffs are present when the next access to that page occurs. This next access is signaled by a segmentation fault.

By contrast bar-u is designed for static, iterative, barrier-based programs, and consumer information is distributed globally at barrier synchronizations. This allows producers to have exact knowledge of consumers. Hence, consumers of data wait for updates before leaving barrier operations, allowing the segmentation faults and additional page protection changes to be avoided.

4. Eliminating OS Memory overhead

Figure 6 shows a breakdown of the execution time for each of the applications running the bar-u protocol. Runtime is broken into sigio handling, wait time, operating system overhead, and application computation. Sigio handling refers to time spent handling incoming requests. Wait time is the time spent waiting for remote requests to succeed. Since nearly all remote misses have been eliminated, this refers primarily to slaves waiting for barrier release messages. Operating system overhead consists of time spent in operating system traps, such as ‘send’, ‘recv’, and ‘mprotect’, which is used to change page protections. Application computation is time spent doing useful work.

Note two things. First, several of the applications (fft, shallow, and swm) have substantial OS components. Although this graph doesn’t break OS overhead down into contributions from individual traps, the majority of this time is spent in mprotect calls. This is an order of magnitude more time than implied by the mprotect time given in Section 3.2.

Second, the efficiency implied by the “app” component of several of the applications does not gibe well with the corresponding speedups. For instance, swm spends 41.7% of the time doing useful work, implying that speedup should be near 8 * 41.7% = 3.3, assuming that the parallel version does no more work than the sequential version. However, the actual speedup is closer to 1.8.

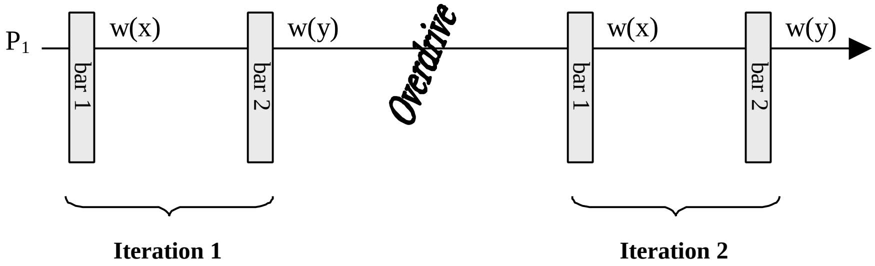

Figure 7: Iterations and Overdrive

We theorized that these discrepancies are caused by a degradation of the operating system performance when under stress. In this case, the stress is an application (CVM) that uses memory in unorthodox ways, i.e. modifying page protections in large address spaces in an unpredictable order. In order to test our theory, we decided to modify bar-u in order to minimize operating system traps that manipulate the application's address space. We could presumably make CVM's behavior appear more orthodox if we ceased manipulating page protections and ceased using segmentation violations (segvs) to detect invalid accesses. Once CVM's behavior fit the standard envelope, we would expect CVM's performance to scale much better.

4.1 Bar-s

Eliminating these two mechanisms one at a time, we first address the use of segvs. Segvs are used for *write trapping,* the trapping of inappropriate accesses to pages. These can be either write accesses to pages that are readable but not writable, or any type of access to invalid pages. If segvs are not used, some other method of write trapping must be used. Given the repetitive nature of our applications, the obvious choice is to use historical behavior to infer future behavior. We call the protocol that uses this technique *bar-s.*

After gathering information for some period of time, bar-s goes into *overdrive* mode, and uses another method of write trapping. This is analogous to what the update protocol already does via copysets. However, copysets are indiscriminate in that they provide no information on how shared accesses relate to synchronization.

For example, Figure 7 shows two iterations of a single process in a parallel CVM run. Each iteration is composed of two barrier epochs: that of barrier 1, and that of barrier 2. After barrier 1, p_1 modifies data *x*. After barrier 2, p_1 modifies *y*. The application goes into overdrive mode after the first iteration. Hence, given the behavior during the first iteration, we expect *x* to be modified after the next occurrence of barrier 1, and *y* to be modified after the next occurrence of barrier 2. Bar-u would trap this write by making a twin when the first write to the page during the barrier epoch occurs. However, without recourse to segvs, we must assume that the write will take place and make the twin ahead of time. We therefore make a twin of *x* and make it writable *before* we leave barrier 1. Upon arriving at barrier 2 for the second time, the diff is created as normal and write protection is removed. The only difference from bar-u at this point is that we have no idea whether the write actually oc-

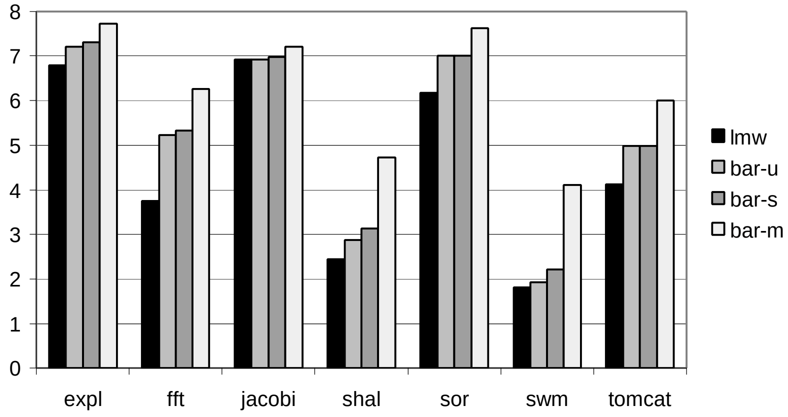

Figure 8: Overdrive Speedups

curred. We can dispense with updating the page elsewhere if the resulting diff is zero-length. However, the twin and diff creations are pure overhead if the write did not happen. We do not expect this to be common, however, as this technique is only applicable when access patterns are highly predictable.

Note that although segvs no longer occur and that we may attempt to create diffs for pages that are not accessed, bar-s will still be correct. Any unanticipated write will be trapped by a segv, at which point the protocol reverts to bar-u.

5. Bar-m

The next step is to eliminate the mprotects. We call the resulting protocol bar-m. Bar-m is identical to bar-s except that we also eliminate mprotect calls once overdrive mode has started. This means that any page that will be written locally while overdrive is in effect must be made writable *before* overdrive goes into effect. Hence, the set of writable pages at any given point during the overdrive portion of an execution can be a strict superset of the pages that the protocol expects to be modified. In Figure 7, for example, the pages of both *x* and *y* must be made writable before overdrive goes into effect. This means that if an application's sharing pattern diverges during overdrive mode (i.e. P_1 modifies *y* after barrier 1), bar-m is not guaranteed to detect the access at the end of the epoch in which it happens. Bar-m is therefore not guaranteed to maintain consistency.

5.1 Results

Figure 8 shows the speedups of bar-s and bar-m for seven of our eight applications. The best speedup from the two lmw protocols and bar-u's speedup are shown for comparison. Barnes is not shown because its sharing pattern, although iterative, is highly dynamic. Work is allocated via non-deterministic traversals of a shared tree structure, resulting in slightly different sharing patterns each iteration. The other seven applications ran correctly under bar-s and bar-m without modification.

		forward	backward	forward by 2	random
266 Mhz Pentium II,	protection	2.0	2.0	2.0	2.4
Linux 2.0.32	signal handling	13.9	13.9	17.6	17.9
66.7 Mhz POWER2,	protection	17.1	**85.1**	17.8	**789.8**
AIX 4.2	signal handling	374.9	267.4	268.6	273.5
275 Mhz 21064A,	protection	20.4	19.0	21.6	24.8
Alpha, Digital Unix 4.0	signal handling	133.0	148.4	130.7	135.2

Table 3: Memory system operations (*u*secs)

Figure 8 shows that bar-s improves upon bar-u by only an average of 2%, indicating that segv handling is not a major source of overhead. However, bar-m achieves a 34% gain on top of the 19% gain from bar-u to the lmw protocols. Since the shared access behavior of the applications is invariant across iterations, bar-u, bar-s and bar-m send exactly the same number of messages and communicate the same amount of data. Hence the difference is entirely due to the lack of memory system interaction via mprotect calls.

5.2 Cost of page protection primitives

Table 3 shows the cost of the page-protection primitives on three platforms. We measured the cost of ‘mprotect’, which is used to change page protections, and *segv* signal handlers. The segv handler is used to detect inappropriate types of accesses to shared pages. For example, when a page is locally invalid, the page is both write- and read-protected. The first subsequent access to the page causes a segv signal handler to be called.

We measured each quantity by marching through a 1000-page data segment in four ways: forward, backwards, forward in increments of two, and in a random order. Table 3 shows that the cost of signal handling remains relatively constant across the different types of progressions for each platform, but Linux handles sequential faults (forward or backward) better than other patterns.

The protection costs, however, explain much of the performance advantage of bar-m on the SP-2. The average cost of a protection change increases by a factor of five when going backward, and more than an order of magnitude when the accesses are in a random order. Note that the pattern of accesses in the real applications is likely to be somewhere between the forward sequential pattern and random.

A possible explanation for AIX’s erratic performance is that the virtual-memory map might be implemented with a single-linked list, augmented with a saved pointer to the last accessed entry. This scheme would explain the good performance for forward traversals of pages and the poor performance of other access orders. The map usually consists of only a few blocks for most processes. However, the page-level protection changes used by a DSM system tend to fragment the list, resulting in poor access times by code that is not optimized for handling many blocks. Linux, the only system for which we have source, uses AVL trees to represent the map. An even better solution would be to combine a fast tree algorithm with block coalescing. Page protections are often manipulated one-by-one during initialization phases, leading to fragmentation of the map. However, portions of “shared” data segments are often accessed by only a single processor. Performance could be improved by coalescing contiguous ranges of such pages back into a single block.

5.3 Discussion

While protocols like bar-m are not likely to be the first option for many users of software DSMs, they could be useful in specific circumstances. Their primary application domain would likely be large, iterative scientific codes, like most of those discussed in this paper. Such applications tend to stretch operating systems in ways that are unexpected for the operating system developers. However, eliminating interrupts and kernel traps will always improve performance even if operating system support is tuned for DSM-like consistency actions.

Nonetheless, protocols such as bar-m should clearly be used only if the programmer *knows*, by whatever means, that the program's data accesses are completely predictable. While running bar-s over similar data sets several times can give some measure of assurance, a clean run of bar-s is by no means proof of a program's repeatability. In this sense, the problem of knowing whether an application is safe for bar-m is analogous to the problem of detecting data races at run-time [9, 10].

One potential source of information is the programmer. While complicated applications could be analyzed by programmers manually, it is more likely that the programmer could verify that the sharing patterns of an abstract algorithm are invariant. Given knowledge that the application closely models the underlying algorithm, the programmer would then have some measure of confidence that the application would execute correctly under a bar-m.

Nonetheless, compilers are a more trustworthy source of information, whether they be parallelizing compilers or compilers for explicitly parallel languages. The information needed to determine that sharing behavior is invariant is clearly a subset of the information that is needed in order to determine precisely what that sharing behavior is. We therefore expect that this information could be obtained more easily and therefore could be more commonly implemented. Moreover, there is likely to be a large class of applications for which determining invariance of access patterns is possible, but determining the nature of the patterns is not.

6. Conclusions

This paper has presented the design and performance of several new protocols that support iterative, parallel applications with stable sharing patterns. We first described lmw-u, an update-based version of a conventional lazy release consistent protocol that improves performance over lmw-i by eliminating most remote misses.

We then described bar-u, a modified *home-based* [15] protocol that performs even better than LRC protocols for this type of application. The reasons include “home” effects, and the fact that home-based protocols are much less complex. Whereas “home-less” LRC protocols can perform poorly for applications that modify (and communicate) large amounts of data, home-based protocols incur less system overhead because they maintain relatively little state, and such state has very short lifetimes. The main drawbacks of home-based protocols are related to problems adapting to dynamic sharing patterns, precisely the sort of pattern that the applications we are investigating do not have.

Finally, we presented and analyzed bar-s and bar-m, two protocols that successively strip away all reliance on kernel protection manipulation. The result is that the application's behavior conforms more closely to the type of behavior expected (and optimized for) by the underlying operating system, and therefore performs much better. Overall, our update home-based protocols average 51% better than the original lmw invalidate protocols for our environment.

7. Bibliography

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

[2] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory," in *Proceedings of the Principles and Practice of Parallel Programming*, 1997.

[3] S. Chandra and J. R. Larus, "Optimizing Communication in HPF Programs on Find-Grain Distributed Shared Memory," in *Proceedings of the 6th Symposium on Principles and Practice of Parallel Programming*, 1997.

[4] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, "Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers," in *Proceedings of the International Parallel Processing Symposium*, 1997.

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

[6] 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.

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

[8] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, pp. 690--691, September 1979.

[9] D. Perkovic and P. Keleher, "Online Data-Race Detection via Coherency Guarantees," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs," in *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[11] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain Access Control for Distributed Shared Memory," in *The Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[12] C.-W. Tseng and P. Keleher, "Enhancing Software DSM for Compiler-Parallelized Applications," in *11th International Parallel Processing Symposium*, 1997.

[13] 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.

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

[15] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October, 1996.