Adaptive Protocols for Software Distributed Shared Memory

Cristiana Amza†, Alan L. Cox†, Sandhya Dwarkadas‡,
Li-Jie Jin†, Karthick Rajamani¶, and Willy Zwaenepoel†

† Department of Computer Science, Rice University
‡ Department of Computer Science, University of Rochester
¶ Department of Electrical and Computer Engineering, Rice University

Abstract

We demonstrate the benefits of software shared memory protocols that adapt at run-time to the memory access patterns observed in the applications. This adaptation is automatic — no user annotations are required — and does not rely on compiler support or special hardware. We investigate adaptation between single- and multiple-writer protocols, dynamic aggregation of pages into a larger transfer unit, and adaptation between invalidate and update. Our results indicate that adaptation between single- and multiple-writer and dynamic page aggregation are clearly beneficial. The results for the adaptation between invalidate and update are less compelling, showing at best gains similar to the dynamic aggregation adaptation and at worst serious performance deterioration.

I. Introduction

Many different protocols have been proposed for implementing a software shared memory abstraction on distributed memory hardware. The relative performance of these protocols is application-dependent: the memory access patterns of the application determine which protocols exhibit good performance. It is therefore appealing to build a system with multiple protocols, and let the system choose between the different protocols based on the access patterns it observes in the application. In this paper we present the design of such an *adaptive* software distributed shared memory system and evaluate its performance.

Specifically, this paper focuses on protocols that implement the lazy release consistency (LRC) memory model [16]. We furthermore assume that shared memory accesses are detected using virtual memory protection. This paper explores the benefits of LRC protocols that adapt to the memory access patterns of the applications, by comparing their performance to non-adaptive versions of the protocols. In particular, we investigate:

1. adaptation between single- and multiple-writer protocols, including adaptation to migratory access patterns,
2. dynamic aggregation of pages into larger transfer units, and
3. adaptation between invalidate and update protocols.

The adaptations considered in this paper are triggered *automatically*: the run-time system detects certain access patterns and switches between protocols accordingly. This automated adaptation distinguishes our work from so called multi-protocol software shared memory implementations (e.g. [8]), in which the user has to annotate the program to select the appropriate protocol. In our experience, removing the need for annotation leads to much improved usability.

The adaptive protocols were implemented in TreadMarks [1]. Our experimental platform is a switched 100Mbps Ethernet consisting of eight 166Mhz Pentium Pro machines running FreeBSD. We use eight applications to demonstrate the performance of the adaptive protocols: 3D-FFT, CG, MG and IS from the NAS benchmark suite [4], Water and Barnes-Hut from the SPLASH benchmark suite [22], Gauss from the TreadMarks distribution, and ILINK from the FASTLINK package [21]. The results indicate that:

1. Adaptation between single- and multiple-writer and dynamic aggregation perform well, in some cases showing substantial performance improvement, and never decreasing performance.
2. Adaptation between invalidate and update is less successful, with performance improvements that match dynamic aggregation in some cases and substantial performance losses in others.

The outline of the rest of this paper is as follows. Section II presents the necessary background information about LRC. Section III presents the possible protocol choices for implementing LRC, and the policies and mechanisms by which the adaptive protocols choose between their alternatives. Section IV describes the experimental environment. Section V describes the applications used. Section VI presents the results of the performance comparison. Section VII discusses related work. Section VIII presents our conclusions.

II. Programming Model

We assume an explicitly parallel programming model, with primitives for process creation and destruction, synchronization, and shared memory allocation and deallocation. Synchronization primitives include mutual exclusion locks and barriers. Shared memory is accessed through load and store instructions. The memory *consistency* model presented to the user is release consistency (RC), a relaxed memory model [13].

In RC, *ordinary* shared memory accesses are distinguished from *synchronization* accesses, with the latter category subdivided into *acquire* and *release* accesses. Lock synchronization maps onto acquires and releases in the obvious way: a `lock` operation corresponds to an acquire, and an `unlock` corresponds to a release. With barriers, a barrier arrival corresponds to a release, whereas a barrier departure corresponds to an acquire. Roughly speaking, RC requires that, before a release by a processor p becomes visible to another processor q, all ordinary shared memory modifications by processor p become visible to processor q. The Lazy Release Consistency (LRC) algorithm [16], one of the possible RC implementations, delays the propagation of shared memory modifications by processor p to processor q until q executes an acquire corresponding to a release by p.

Programs without data races, i.e., programs with sufficient synchronization such that any pair of conflicting memory accesses is separated by a release-acquire pair, produce the same results on an RC or an LRC memory system as on a conventional sequentially-consistent memory system [17]. Performance, however, can be much improved by the use of RC or LRC, especially for software implementations of shared memory, because the messages propagating the shared memory modifications can be delayed and coalesced with the synchronization messages, leading to a substantial reduction in communication [8], [16]. In addition to being data-race-free, all synchronization in the program must be done through the primitives supplied by the runtime system, so that it can take the required consistency actions at synchronization points.

We assume that the shared memory is implemented as a global virtual memory segment, shared by all processors. The virtual memory protection hardware is used to detect access to individual pages. Some of these accesses may cause page faults, which then trigger protocol operations as described in the next section.

III. Protocols

A. Single- vs. Multiple-Writer Protocols

A.1 The Basic Protocols and Their Tradeoffs

In a single-writer protocol, there is a single writable copy of a page at any given time [14]. The processor currently holding the writable copy of a page is called the *owner* of that page. Several read-only copies of the page may coexist with the writable owner copy. According to the definition of RC, a read-only copy may be temporarily inconsistent with the writable copy, but it must be brought up-to-date when the processor on which it resides synchronizes with the owner. Assume, for instance, that an invalidate protocol is used and synchronization is by means of a barrier. If the owner has modified a particular page, it creates an *owner write notice* for that page, containing its processor id and a version number. The barrier protocol causes the owner write notice to be transmitted to the processors which have read-only copies of the page, and these processors then invalidate their copies. On a subsequent access miss, they retrieve the page from the owner. The owner write-protects his copy during the first retrieval. Before a processor may write on a page, it must obtain ownership from the current owner. The owner is located by means of the write notice with the highest version number, possibly by forwarding if ownership has changed since this write notice was received. Once ownership is obtained, the page's version number is incremented by one.

In contrast, in a multiple-writer protocol, there may be several writable copies of a page on different processors [8]. Each processor with a writable copy records its own modifications to the page by a technique called *twinning and diffing*. Pages are initially write-protected so that the first write access to a page causes a protection violation. At this point, the system makes a copy of the page, the twin, and unprotects the original page. To detect what modifications have been made to a page, the current copy is compared word-by-word to the twin, and a record of the modifications, the diff, is constructed. Continuing the above example, when an invalidate protocol is used and synchronization is by means of barriers, each processor that has modified a page constructs a *write notice* for that page, which is forwarded by the barrier protocol to all processors with copies of that page.[1] A processor might receive several write notices for a single page. These write notices cause the page to be invalidated. On an access miss, the diffs corresponding to these write notices have to be retrieved and applied to the processor's current copy of the page.

The tradeoff between single- and multiple-writer protocols is dependent on the access pattern to the page, and affects both execution time and memory overhead. If multiple processors write concurrently to different parts of a page (write-write false sharing), then multiple-writer protocols achieve better performance, because they do not incur the cost of transferring the page over the network to the next writer. Even if there is only a single writer, it may be advantageous to use twinning and diffing. This scenario occurs when the writer modifies only a small portion of the page. The multiple-writer protocol transmits only those modifications, while a single-writer protocol transmits the entire page.

If, however, only a single processor writes to a page at any given time, and this processor modifies a large part of the page, then the single-writer protocol avoids the cost of twinning, diffing, and diff application, without much increase in communication. More importantly, it avoids a pitfall of the multiple-writer protocol, called *diff accumulation* [18], a scenario in which a number of partially or completely overlapping diffs are transmitted, significantly increasing the amount of communication. While it is possible to modify the multiple-writer protocol to eliminate the overlap, there is a high computational cost to pruning useless data from older diffs each time a new diff is created. It is more efficient to manage the page in single-writer mode.

Finally, while the memory overhead for the single-writer protocol is negligible, the multiple-writer protocol has to allocate memory for the twins and the diffs. This extra overhead may cause an application to page to disk with a multiple-writer protocol, while running in memory with a

[1] The information in the write notices of the multiple-writer protocol is more complicated than the version number present in the owner write notices of the single-writer protocol. In particular, it contains a *vector timestamp* that allows the write notice to be partially ordered w.r.t. write notices from other processors.

single-writer protocol.

A.2 Adapting between Single- and Multiple-Writers

In the adaptive protocol used in this paper, all pages start out in multiple-writer mode. A page may switch to single-writer mode by one of two events:
1. A processor receives a diff request for a page, and it has modified the entire page. In this case, the page is clearly single-writer, and there is no reduction in communication by sending a diff.
2. A processor sends out diff requests for a page, it receives no concurrent diffs, and the sum of the sizes of the diffs received is bigger than the page size. This is indicative of the diff accumulation phenomenon discussed earlier. Since there are no concurrent diffs, there is no write-write false sharing. Looking ahead to the time where a different processor requests the diffs for this page, keeping the page in multiple-writer mode would cause more data to be sent than a page. It is therefore more efficient to put the page in single-writer mode.

A page may switch back to multiple-writer mode at the onset of write-write false sharing, which is detected by the *ownership refusal* protocol, a modification to the single-writer protocol for locating and transferring ownership [2]. On a release (an unlock or a barrier arrival), a processor communicates both its owner write notices and its multiple-writer write notices. On a write fault to a page in single-writer mode, a processor requests ownership, as in the single-writer protocol: The owner is located, using the owner write notice with the highest version number. This version number is included in the ownership request message. If the recipient of the message is no longer the owner, or if the version number has changed, write-write false sharing has been detected, the ownership request is *refused*, and the page is put into multiple-writer mode. Otherwise, ownership is *granted*, the old owner write protects its copy of the page, the requester becomes the new owner, the version number is incremented, and the page stays in single-writer mode.

The essential aspect that needs to be understood about this protocol is that it correctly detects the presence or absence of write-write false sharing. Consider the example of a data item protected by a lock, and assume that there is no write-write false sharing on the page containing that data item. When processor p acquires the lock, it receives the owner write notice from the previous owner q with version number V. When p writes on the page, it incurs a page fault, and it tries to achieve ownership. It sends an ownership message to q, including the version number V. By our assumption that there is no write-write false sharing on the page, no other processor has attempted to write on the page, and therefore q is still the owner and the page version's number is still V. Therefore, the ownership is granted, and p becomes the new owner. Consider next the case where there is write-write false sharing on the page, either because q or some other processor wrote on a different part of the page. If q wrote to the page, it must have re-acquired ownership of the page, and thus it must have incremented the version number. If a different processor wrote to the page, it must have acquired ownership, and q is no longer the owner. In either case, p's ownership request is refused, and the page is put in multiple-writer mode. For a more detailed description and a correctness argument, we refer the reader to Amza et al. [2].

A.3 Adapting to Migratory Access

Adaptation to migratory access only makes sense in the context of an adaptive protocol operating in single-writer mode (or a single-writer protocol) where its purpose is to eliminate the need for explicit ownership messages. Compared to the base multiple-writer protocol, the adaptive protocol requires an extra message to acquire ownership in the following scenario. A processor takes a read fault on an invalid page, obtains the diffs to validate the page, and then later takes a write fault on the page. With the base multiple-writer protocol, a twin is created but no messages are sent at the time of the write fault. With the adaptive protocol, an ownership request is sent. The scenario described is that of a *migratory* access pattern: a sequence of reads followed by a sequence of writes by one processor with no intervening accesses by other processors [26].

Detecting migratory access and eliminating the explicit ownership message is straightforward [9], [24]. If a page is migratory, when a processor performs its first read from the page, it will fault because the page is invalid. Its request for the page will go to the processor that still owns the page. If that processor accessed the page in a similar, migratory fashion, it will preemptively send ownership along with the page. Later, if the page changes access pattern, for example, to producer/consumer, the overhead to switch will be one ownership request.

B. Adaptive Run-time Aggregation of Pages

B.1 The Basic Protocols and Their Tradeoffs

Software DSM systems based on virtual memory techniques traditionally use the hardware page as the unit of access detection and as the unit of transfer. The single-writer, multiple-writer, and adaptive protocols discussed in Section III-A all follow this approach. Depending on whether a single- or multiple-writer protocol is used, a diff or a whole page is transferred, but in both cases, access detection is done on a per-page basis, and the data transferred in a page fault response always pertains to a single hardware page. For simplicity, the discussion in this section is cast in terms of the multiple-writer protocol, unless otherwise noted, but it can easily be extended to the single-writer protocol and the adaptive single-writer/multiple-writer protocol described in Section III-A.

Both the unit of access detection and the unit of transfer can be increased, for instance by using a multiple of the hardware page size. Doing so trades off *aggregation* vs. the potential for increased false sharing. Aggregation reduces the number of messages exchanged. If a processor accesses several pages in succession, a single page fault request and reply now suffice, where before multiple exchanges were required. As a secondary benefit, the number of page faults is also reduced. These gains, however, come at the expense of potentially increased false sharing. False sharing may lead to an increase in the amount of data exchanged. Assume, for instance, that processor p writes to successive pages a and b, and processor q accesses only a. With the base page size, only the diffs for a are transferred, but if the page size is doubled, the diffs for a and b are transferred. Worse, false sharing may also lead to an increase in the number of messages. If processor p writes a, processor q writes b, and processor r reads a, two message exchanges occur with a doubled page size, one between p and r, and one between q and r, where an exchange between p and r sufficed with the base page size. The effects of false sharing are aggravated under the single-writer protocol, causing more and larger page transfers. Under the adaptive single-writer/multiple-writer protocol the larger page may be put in multiple-writer mode, while the individual hardware pages could have been handled in single-writer mode.

B.2 The Adaptive Protocol

In this section, we present a protocol that continues to use the hardware page as the unit of detection, but adaptively coalesces pages into *page groups* for the purpose of transfer. The algorithm monitors the access patterns on each processor, and tries to construct page groups so as to increase aggregation without incurring the harmful effects of false sharing.

The diffs for *all* of the pages in a group are requested at the first fault on any page that is a member of the group. Requests addressed to the same processor are combined into one message, resulting in fewer request messages and enabling the data transfer to occur in one message as well. Even if the diffs must come from different processors, there is still an advantage to requesting the diffs for all pages in the group at once, because those processors can return the diffs in parallel rather than in sequence.

A processor uses two different mechanisms for grouping pages. The first mechanism is based on the past accesses on that processor itself. Essentially, the processor groups pages that were accessed during the previous synchronization interval. In order to avoid packet loss in the network, the implementation limits the maximum number of pages in a single group to eight. Thus, more than one group may be formed at a synchronization point. If two or more groups are formed, the pages are assigned to groups in the order they were accessed. The second mechanism is based on past accesses of other processors. It comes into play only if the first mechanism did not produce a group for the missing page. The faulting processor checks if the page was modified by a single processor during the previous synchronization interval, and, if so, it requests from that processor any contiguous pages that were modified during that interval. Again, the number of pages in any group is limited to eight.

In order to allow the membership of a group to change over time, the algorithm keeps every page invalid until the first access to that page occurs. Thus, a page may be kept invalid, even though it has been updated by an access to another page within the same group. When the page fault handler is triggered by an access to such a page, it can simply change the page's state to valid without requesting any data. In this case, the page will remain a part of its group. If, however, the page is never accessed, it will be dropped from the group at the next synchronization point. Hence, this strategy allows the algorithm to adapt to any change in the program's access pattern over the course of its execution.

C. *Invalidate vs. Update*

C.1 The Basic Protocols and Their Tradeoffs

In an invalidate protocol, a page is invalidated when the processor becomes aware of a remote modification. In LRC, this happens at the time of a synchronization. A synchronization message, for instance, a lock grant or a barrier departure message, contains a number of (owner) write notices, indicating which pages have been modified. When the processor later accesses one of these pages, it incurs an access miss. Depending on whether a single- or multiple-writer protocol is in use, either the whole page or the diffs are fetched. In an update protocol, instead, the modifications to the page are sent with the synchronization message. Pages are never invalidated.

The tradeoffs between invalidate and update protocols are well known [12]. Update protocols send substantially more data, including data that the processor may never access or that may be overwritten by newer data before the processor accesses the data originally sent. Invalidate protocols only retrieve the data for the pages the processor accesses, but they pay the penalty of the access miss fault and the round-trip latency to get the modifications. In addition, in release-consistent software DSM, update protocols naturally include aggregation: when a processor modifies several pages, all the modifications are sent in a single message to the other processor(s).

C.2 The Adaptive Protocol

The adaptive invalidate/update protocol updates the pages that the processor is expected to access and invalidates the other pages. As with the aggregation for invalidate protocols described in Section III-B, we limit a single update message to contain data for no more than eight pages in order to avoid packet loss in the network. Prediction of future accesses may be done in a variety of ways. For programs based on barriers, each processor p records the set of processors from which it receives a page fault request for a particular page. When p arrives at the next barrier, if it has modified a particular page, it sends updates for that page to the processors in the set it has computed during the interval before the barrier [15]. These processors return negative acknowledgements to these updates, if they receive a second update for a page and have not accessed the page since the first update. For data protected by a lock, we use the method proposed by, among others, Monnerat and Bianchini [19], and Speight and Bennett [23]. We

Application	Data size	Sync.	Time (sec.)
Water	512 molecules	b,l	56.8
Barnes	32K bodies	b	191.7
IS	21×15	b	5.1
3D-FFT	128×64×64	b	51.7
MG	128×128×128	b	838.9
CG	14,000×14,000 (sparse)	b	113.7
Gauss	1024×1024	b	34.4
ILINK	CLP	b	776.8

TABLE I

Applications, input data sets, synchronization (l=locks, b=barriers), and sequential execution time

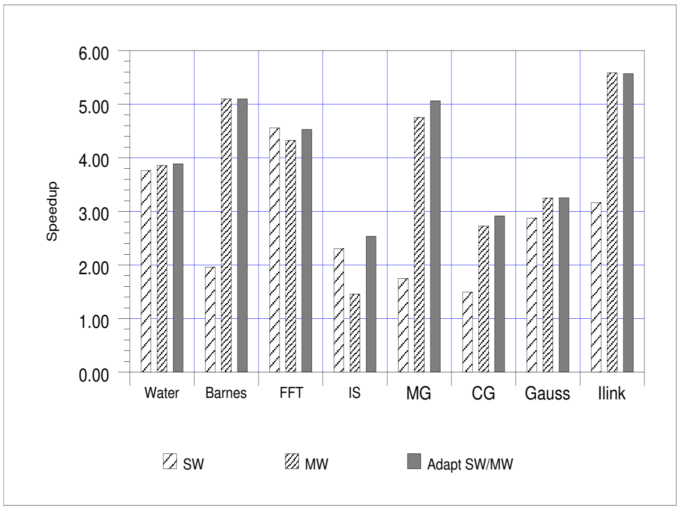

Fig. 1. Speedup comparison: single-writer, multiple-writer, and adaptive protocols.

record which pages a processor modifies while it holds the lock. Updates for these pages are sent to the next acquirer of the lock, while any other modified pages are invalidated.

IV. Experimental Environment

Our experimental platform is a network of eight 166MHz Pentium Pros running FreeBSD 2.2.5. Each machine has a 256K byte secondary cache and a 64M byte memory. The hardware page size is 4K bytes. The network connecting the machines is a switched, full-duplex 100Mbps Ethernet.

TreadMarks uses the UDP/IP protocol for interprocessor communication. The round-trip latency for a 1-byte message using the UDP/IP protocol is 196 microseconds on this platform. The time to acquire a lock varies from 256 to 393 microseconds. The time for an eight processor barrier is 481 microseconds. The time to obtain a diff varies from 387 to 1,225 microseconds.

V. Applications

We use eight applications in this study. Water and Barnes-Hut come from the SPLASH benchmark suite [22]. Integer Sort (IS), 3D-FFT, Multigrid (MG) and Conjugate Gradient (CG) come from the NAS benchmark suite [4]. Gauss is a Gaussian elimination kernel distributed with TreadMarks. ILINK is part of the FASTLINK package [21] of genetic linkage analysis programs.

Table I summarizes the relevant characteristics of the applications. It includes for each application, the data set size used, the method of synchronization (locks, barriers, or both), and the sequential running times. Sequential running times were obtained by removing all synchronization from the TreadMarks programs; these times were used as the basis for the speedup figures reported later in the paper.

VI. Results

For each of the applications we show speedups under the following scenarios:

1. the single- and multiple-writer protocols and the adaptive single-writer/multiple-writer protocol,
2. the adaptive single-writer/multiple-writer protocol plus dynamic aggregation, and
3. the adaptive single-writer/multiple-writer protocol plus invalidate/update adaptation, including aggregation of the updates.

The effects of dynamic aggregation are independent of whether the base protocol is the single-writer, multiple-writer, or adaptive single-writer/multiple-writer protocol. Hence, we only present the results for dynamic aggregation using the base protocol with the best overall performance, the adaptive single-writer/multiple-writer protocol.

Similarly, the effects of adaptation between invalidate and update are the same for the single-writer, multiple-writer, and adaptive single-writer/multiple-writer protocol. Furthermore, since the "update" part of the adaptive invalidate/update protocol inherently includes aggregation, and since aggregation is always beneficial with invalidate protocols, we compare the invalidate-based adaptive single-writer/multiple-writer protocol with aggregation to the adaptive invalidate/update, single-writer/multiple-writer protocol.

A. *Single- vs. Multiple-Writer Protocol*

Figure 1 shows the speedup on eight processors for each of the applications using the single-writer protocol, the multiple-writer protocol, and the protocol that adapts between the two, including the adaptation to migratory accesses. An invalidate protocol using the hardware page size is used, as in the base TreadMarks system.

We first compare the non-adaptive single- and multiple-writer protocols. As expected, the amount of write-write false sharing determines the tradeoff. The single-writer protocol performs better than the multiple-writer protocol on applications with no write-write false sharing and large overlapping diffs (IS), performs comparably on applications with low write-write false sharing (Water, 3D-FFT, Gauss), and worse for applications with high write-write false sharing (Barnes, MG, CG, and ILINK). Comparing

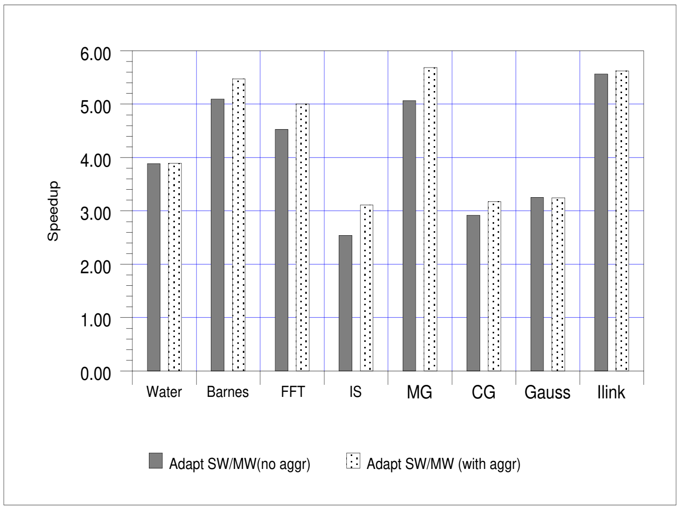

Fig. 2. Speedup comparison: protocols with and without dynamic aggregation.

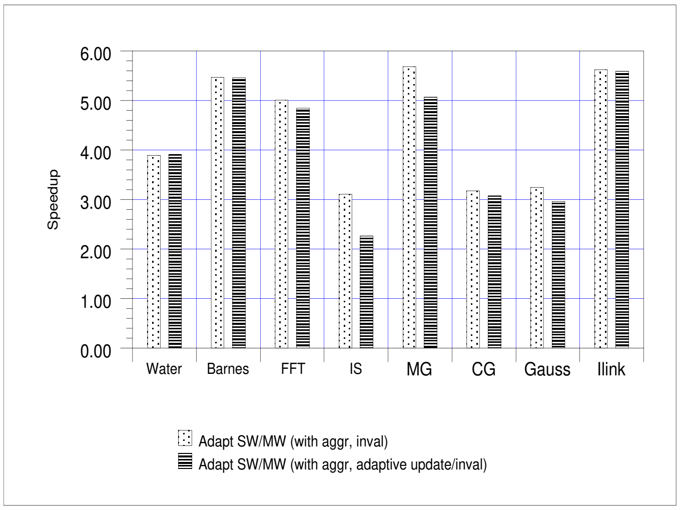

Fig. 3. Speedup comparison: invalidate and adaptive invalidate/update protocols.

the adaptive to the non-adaptive protocols, we see from Figure 1 that the adaptive protocol matches or exceeds the speedup of the best of the non-adaptive protocols.

The adaptation that optimizes migratory access only affects IS. None of the other programs, such as Water, that have migratory data modify the entire page or suffer from significant diff accumulation. Consequently, they do not switch to single-writer mode, and thus the migratory optimization is not needed. For IS, it limits the ownership messages to one per page per iteration, instead of eight.

We do not present the memory demands for the protocols here, but, we offer the following anecdote: for a larger 3D-FFT data set ($256 \times 128 \times 128$), the single-writer and adaptive protocols performed well, running completely in main memory, while the multiple-writer protocol paged because of the twins and diffs it stored, causing a 15-fold increase in execution time. (See Amza et al. [2] for a detailed account.)

B. *Dynamic Aggregation of Pages*

Figure 2 shows the speedups achieved with dynamic page aggregation, in addition to adapting between single- and multiple-writer and adapting to migratory access. As a baseline for comparison, we reiterate in Figure 2 the speedups from Figure 1 for the adaptive single-writer/multiple-writer protocol. Five out of the eight applications benefit from dynamic page aggregation: Barnes-Hut, 3D-FFT, IS, MG, and CG. The benefits for IS derive from the aggregation based on write accesses by other processors, while the benefits for the other four applications derive from the past access patterns by that processor itself. In IS, which sees the greatest benefits, processors exchange a large amount of data, leading to a significant reduction in the number of messages, with the attendant performance benefits. A similar argument explains the somewhat smaller improvements for Barnes-Hut, 3D-FFT, CG and MG.

Surprisingly, three of the applications that benefit from aggregation, Barnes-Hut, MG, and CG, suffer from write-write false sharing. This illustrates the fact that dynamic page aggregation can reduce the number of messages without increasing false sharing.

C. *Invalidate vs. Update*

Figure 3 shows the speedups for the protocol that adapts between invalidate and update (including aggregation of the updates), adapts between single- and multiple-writer, and adapts to migratory access. The results are shown along side those for the invalidate protocol that performs dynamic page aggregation from Figure 2.

The benefits of automatic adaptation between invalidate and update are questionable when such a protocol is compared to a base protocol that performs aggregation. Its benefits are limited to avoiding page faults and round-trip latencies, resulting in a small improvement. In many cases, these improvements are offset by the additional data transfer. Typically, the additional data transfer comes from changes in the sharing pattern. For example, IS consists of a number of iterations, each of which is divided into a number of migratory phases followed by a phase in which the data produced by any single processor is consumed by all other processors. This latter phase causes the adaptive algorithm to send updates to all processors in the first migratory phase of the next iteration. The negative acknowledgements halt these updates after two migratory phases, but the large amount of unnecessary data sent in these two phases causes performance to deteriorate substantially. Adaptation between invalidate and update is, however, attractive in some cases if the invalidate mode of the base protocol does not support aggregation.

VII. Related Work

A large number of software shared memory systems have been built (e.g. [5], [8], [14], [20], [25], [27]). Although the work described here is done in the context of a specific system, TreadMarks, many of the ideas are applicable to other systems as well. First, the adaptation between single- and multiple-writer protocols carries over to all page-based systems. Second, aggregation should prove to be beneficial to all systems, especially the ones that use smaller consistency units. Finally, the tradeoff between update and invalidate also applies to these other systems, although the nature of the tradeoff may change substantially if compiler support is used to determine the choice between update and invalidate [5], [11].

The multiple-writer protocol described in this paper is the one in use with the current version of TreadMarks [1]. The single-writer protocol is a variation of the one presented by Keleher [14]. The adaptive single-writer/multiple-writer protocol extends our earlier work [2] on this topic. In this earlier work, we chose a protocol that started out in single-writer mode, because of its reduced memory use (no twins are ever made for pages that remain in single-writer mode). We found that the same reduction in memory use can be achieved by a protocol that starts out in multiple-writer mode, by not creating the initial twin, which contains all zeroes. Starting in multiple-writer mode allows for a straightforward adaptation according to the size of the diffs.

The adaptive DSM system described by Monnerat and Bianchini [19] is most closely related to our work. They also investigate the adaptation between single- and multiple-writer protocols, and adaptation between invalidate and update. In their system, pages are classified as migratory, producer/consumer or falsely shared. Single-writer mode is used for migratory and producer/consumer pages, while the falsely shared pages are maintained in multiple-writer mode. Updates are used only for migratory and producer/consumer pages. Keleher et al. [15] and Espeight et al. [23] have also investigated the benefits of allowing a software shared memory system the choice between invalidate and update. However, to the best of our knowledge, all of these studies were conducted in the absence of aggregation for the invalidate protocol, inflating the perceived benefits of update. We have demonstrated that communication aggregation is the key to improving performance in both invalidate and update protocols. Adding dynamic aggregation to the invalidate protocol provides the same benefits as using an update protocol, without the risk of sending extra messages.

Amza et al. [3] investigated the benefits of dynamic page aggregation. They did not, however, combine aggregation with other forms of adaptation. Lu et al. [18] found that aggregation is the main reason that message-passing programs outperform (software) shared-memory programs. Overall, they found that for six out of their eight applications the speedup on TreadMarks was within 85% of that achieved by PVM. With the best static page aggregation for each of those six applications, the speedup on TreadMarks improved to within 95% of the speedup on PVM. These results were obtained on two platforms, one of which, the 155Mbps ATM network of eight SPARCstation-20 Model 61 workstations, is similar to the platform used in this paper.

Our adaptive single-writer/multiple-writer protocol addresses the most extreme cases of a less common problem, diff accumulation, found by Lu et al. [18]. Diff accumulation in IS contributed to the worst performance with respect to PVM. TreadMarks' speedup was only 42% of PVM's. With diff accumulation manually removed, the speedup improved to within 71% of the speedup on PVM. Our adaptive protocol automatically achieves a similar improvement.

Several other systems both hardware and software have investigated configurability or adaptivity as a means of improving performance.

Shasta [20] features configurable consistency units to address the requirements of applications with fine-grain sharing at the expense of higher memory overheads.

Munin [8] uses multiple protocols to handle data with different access characteristics. The novelty in our work is that it chooses automatically between different protocols. In Munin, the choice of protocol was based on somewhat burdensome user annotations.

Cashmere [25] improves on the home-based protocol introduced by Zhou et al. [27], allowing dynamic migration of the home node. The home-based protocol allows a single-writer optimization that avoids diffing overhead when the home node is the only writer for the page. The downside is that whole pages are fetched on faults, even if the amount of data modified is small.

Dubnicki and LeBlanc [10] proposed a scheme to reduce the impact on performance due to a mismatch between the cache block size and the sharing patterns exhibited by a given application. They adjusted the amount of data stored in a cache block according to recent reference patterns. They found that the adjustable cache-block-size implementation did better than the best fixed-size implementations for most of the programs in their suite.

The adaptation to migratory behavior was first suggested by Cox and Fowler [9] and Stenstrom et al. [24] in the context of hardware shared memory machines.

Another form of adaptivity that is important in networks of workstations is adapting to environmental characteristics such as processor and network load [6], [7]. This form of adaptivity is orthogonal to the one discussed in this paper.

VIII. Conclusions

We have described software DSM protocols that automatically adapt, on a per-page basis, to the access patterns in the application. The protocols dynamically choose between single- and multiple-writer protocols. Pages can be dynamically aggregated into larger page groups. Finally, the protocols choose dynamically between invalidate and update. All adaptation is automatic.

The choice between the single- and multiple-writer protocols is based on the presence of write-write false sharing and on write granularity. In addition, the protocol detects migratory behavior, and chooses a version of the protocol optimized accordingly. Aggregation uses records of earlier accesses by a processor to coalesce pages into page groups, in the expectation that those pages will be accessed again by the processor. The choice between invalidate and update is based on whether we expect the destination to access the modified data before it is overwritten or not. The three adaptations can easily be combined.

Adaptation between single- and multiple-writer and dynamic aggregation proved to be the most beneficial, never causing any deterioration and providing substantial improvement for some applications. Our automatic adaptation between invalidate and update was less successful, showing at best gains equal to the dynamic aggregation adaptation and at worst serious performance deterioration. We speculate that it may be difficult to find a fully automatic, purely run-time algorithm for adaptation between invalidate and update, and that either compiler or user input may be necessary to achieve good performance.

Acknowledgements

This work was supported in part by NSF grants CCR-9702466, CCR-9705594, CCR-9410457, CCR-9457770, CCR-9502500, CCR-9521735, CDA-9502791, and MIP-9521386, by the Texas TATP program under Grant 003604-017, and by grants from IBM Corporation and from Tech-Sym, Inc.

References

[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*, 29(2):18–28, February 1996.

[2] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 261–271, February 1997.

[3] 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 6th Symposium on the Principles and Practice of Parallel Programming*, pages 90–99, June 1997.

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

[5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, June 1992.

[6] R.D. Blumofe and P.A. Lisiecki. Adaptive and reliable parallel computing on network of workstations. In *Proceedings of the USENIX 1997 Annual Technical Symposium*, January 1997.

[7] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and piranha. *IEEE Computer*, 28(1), January 1995.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[9] A. L. Cox and R.J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[10] C. Dubnicki and T. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–180, May 1992.

[11] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, October 1996.

[12] S.J. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[14] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 91–98, May 1996.

[15] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29:126–141, October 1995.

[16] 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*, pages 13–21, May 1992.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[18] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):56–78, June 1997.

[19] L.R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[20] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[21] A.A. Schäffer. Faster linkage analysis computations for pedigrees with loops or unused alleles. *Human Heredity*, 46(4):226–235, jul 1996.

[22] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[23] W.E. Speight and J.K. Bennett. Using multicast and multithreading to reduce communication in software DSM systems. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[24] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[25] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[26] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[27] 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 Second USENIX Symposium on Operating System Design and Implementation*, pages 75–88, nov 1996.