Software DSM Protocols that Adapt between Single Writer and Multiple Writer

Cristiana Amza[†], Alan L. Cox[†], Sandhya Dwarkadas[‡], and Willy Zwaenepoel[†]

[†] Department of Computer Science
Rice University
{amza, alc, willy}@cs.rice.edu

[‡] Department of Computer Science
University of Rochester
sandhya@cs.rochester.edu

Abstract

We present two software DSM protocols that dynamically *adapt* between a single writer (SW) and a multiple writer (MW) protocol based on the application's *sharing* patterns. The first protocol (WFS) adapts based on write-write false sharing; the second (WFS+WG) based on a combination of write-write false sharing and write granularity. The adaptation is automatic. No user or compiler information is needed. The choice between SW and MW is made on a per-page basis.

We measured the performance of our adaptive protocols on an 8-node SPARC cluster connected by a 155 Mbps ATM network. We used eight applications, covering a broad spectrum in terms of write-write false sharing and write granularity. We compare our adaptive protocols against the MW-only and the SW-only approach. Adaptation to write-write false sharing proves to be the critical performance factor, while adaptation to write granularity plays only a secondary role in our environment and for the applications considered.

Each of the two adaptive protocols matches or exceeds the performance of the best of MW and SW in seven out of the eight applications. For these applications, speedup improvements over SW range from 1.02 to 2.7. The largest improvements over SW occur for applications with high write-write false sharing. Compared to MW, speedups improve by a factor of 1.02 to 1.6, with the largest improvements occurring for applications with little or no write-write false sharing. Both WFS and WFS+WG speedups fall below the best of MW and SW for one application, but only by a factor of 1.09 and 1.06. In addition, memory usage is reduced considerably compared to MW, in some cases making the memory overhead all but negligible.

1 Introduction

This paper focuses on protocols for implementing lazy release-consistent (LRC) [14] software distributed shared memory (DSM) [16] on commodity hardware. Both single writer (SW) [13] and multiple writer (MW) [6] protocols have been used to implement LRC. SW protocols allow only a single writable copy of a page at any given time. Furthermore, they always transfer a whole page to satisfy an access miss. With MW protocols, several writable copies of a page may co-exist. Instead of transferring whole pages, MW protocols transfer *diffs*, records of the modifications made to a page.

SW protocols suffer from the ping-pong effect in the case of write-write false sharing (concurrent writes from different processors to non-overlapping parts of the same page). Furthermore, if only a single word in a page is changed, then it is clearly undesirable to transmit the entire page, especially on a low bandwidth network.

MW protocols solve these two problems, but suffer some drawbacks of their own. First, there is an execution cost to recording and merging changes from multiple writers (i.e., creating and applying the diffs). Much of this overhead is incurred regardless of whether the page has multiple writers or not, adding unnecessary overhead in the case where there is no false sharing. In particular, in the case where an entire page is modified by a single writer, a MW protocol adds a sizable cost without any reduction in communication. Furthermore, there is a significant memory overhead for recording the modifications. The memory costs can be bounded by garbage collection, but frequent garbage collection results in added execution time.

CVM [13] uses a SW protocol, while TreadMarks [3] uses a MW protocol (see the work of Keleher [13] for a study of the tradeoffs). Other systems (such as Munin [6]) allow multiple protocols to be used, but require user annotation to choose between them. In this paper we take an alternative approach. We observe that for some applications a MW protocol is preferred while for others a SW protocol is more desirable. Even within a single application, different pages may be best handled by one protocol or the other. As a result we have designed two adaptive protocols that choose dy-

---

This work is supported in part by the National Science Foundation under Grants CCR-9410457, BIR-9408503, 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.

namically, on a per-page basis, whether to use a SW or a MW protocol. The choice is fully automated, and no user or compiler annotations are required. Instead, the runtime system monitors the shared memory access patterns and decides on the appropriate protocol. We show that this can be done with little overhead, as an extension of the SW and MW protocols. The first adaptive protocol (WFS) adapts to the presence of write-write false sharing. It chooses a MW protocol if there is write-write false sharing on a page and a SW protocol otherwise. The second adaptive protocol (WFS+WG), in addition, takes into account write granularity, and uses diffs for pages with small write granularity, even if they do not exhibit write-write false sharing.

The adaptive protocols were implemented in TreadMarks [3]. Eight applications were used to demonstrate the performance: Red-Black SOR and TSP are small kernels; Barnes-Hut and Water are from the Splash benchmarks suite [19]; IS and 3D-FFT are from the NAS benchmark suite [4]; Shallow is a small weather modeling code from NCAR [18]; and ILINK is a production computational genetics code [11]. These applications cover a wide spectrum in terms of write-write false sharing and write granularity. We present performance results on a 155Mbps ATM network connecting 8 SPARC-20 model 61 workstations. We compare the performance of the adaptive protocols to the non-adaptive MW protocol used in TreadMarks and to a non-adaptive SW protocol similar to the one used in CVM. Each of the two adaptive protocols matches or exceeds the performance of the best of the non-adaptive protocols in seven out of the eight applications. Speedup improvements are as high as a factor of 2.7 over SW for applications with high write-write false sharing and as high as a factor of 1.6 over MW for applications with little or no write-write false sharing. Both WFS and WFS+WG speedups fall below the best of MW and SW for one application, but only by a factor of 1.09 and 1.06, respectively. In addition, memory usage is reduced considerably compared to MW, in some cases making memory cost all but negligible.

The rest of this paper is organized as follows. Section 2 discusses LRC, MW, and SW protocols. Section 3 presents the mechanisms by which the protocols adapt between SW and MW mode. Section 4 describes the experimental environment. Section 5 describes the eight applications used. Section 6 presents the results of the performance comparison. Section 7 discusses related work. Section 8 presents our conclusions.

2 Background

In the following, we introduce LRC, the TreadMarks MW protocol, and our implementation of the CVM SW protocol.

2.1 Lazy Release Consistency

Release consistency (RC) is a relaxed memory consistency model [10]. In RC, *ordinary* shared memory accesses are distinguished from *synchronization* accesses, with the latter category subdivided into *acquire* and *release* accesses. Acquire and release accesses correspond roughly to the conventional synchronization operations on a lock, but other synchronization mechanisms can be built on this model as well. Essentially, RC allows the effects of ordinary shared memory accesses to be delayed until a subsequent release by the same processor is performed.

The Lazy Release Consistency (LRC) algorithm [14] delays the propagation of modifications to a processor until that processor executes an acquire. To do so, LRC uses the *happened-before-1* partial order [2]. The *happened-before-1* partial order is the union of the total processor order of the memory accesses on each individual processor and the partial order of release-acquire pairs. Vector timestamps are used to represent the partial order [14]. When a processor executes an acquire, it sends its current vector timestamp in the acquire message. The last releaser then piggybacks on its response a set of *write notices.* These write notices describe the shared data modifications that precede the acquire according to the partial order. These shared data modifications must be reflected in the acquirer's copy. In this paper we consider invalidate protocols, in which the arrival of a write notice for a page causes the page to be invalidated. On a subsequent access miss to an invalid page, it is made valid by requesting and applying all modifications described by the write notices for that page.

One of the appealing aspects of LRC is that it avoids any ping-pong effect due to read-write false sharing. If one processor writes on one part of a page and another processor reads from another part of the same page, there need not be any communication between the two processors until they subsequently synchronize. Write-write false sharing, however, remains a problem.

2.2 The Multiple Writer Protocol

MW protocols have been developed to address the write-write false sharing problem. With a MW protocol, two or more processors can simultaneously modify their local copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing.

The write notices used in the MW protocol include the processor id and the vector timestamp of the interval during which the page was modified. A faulting processor uses this information to locate and apply the modifications required to update its copy of the page.

In TreadMarks, detection of modifications is done by *twinning* and *diffing.* A page is initially write-protected, so that at the first write a protection violation occurs. TreadMarks then makes a copy of the page (a *twin*), and removes the write protection so that further writes to the page can occur without any software intervention. The twin and the current copy are later compared to create a *diff*, a runlength encoded record of the modifications to the page. These diffs are transmitted in response to requests from faulting processors.

Garbage collection is initiated when the diff space on one or more processors is exhausted. Global synchronization is used to implement garbage collection. All concurrent writers of a page validate their copy of the page by applying all necessary diffs. All other copies of the page and all diffs are deleted.

2.3 The Single Writer Protocol

A SW protocol allows only one writer for a page at any given time. The processor currently holding write privileges to the pages is called the *owner*. Each page has a *version number*, which is incremented every time ownership is acquired (or reacquired by the same processor). When ownership changes, both nodes have the new version number for the page. In response to an acquire request, the owner sends out *owner write notices* for the pages it modified. An owner write notice includes the processor id and the page's version number.

On a write fault, the faulting processor requests ownership of the page. A *static ownership* algorithm is used to locate the owner. This method involves forwarding of requests through a statically assigned home processor. Ownership and the page contents are then sent from the current owner to the requester. On a read fault, there is no transfer of ownership. Instead, the faulting processor p asks for a copy of the page from the processor q named in the owner write notice with the highest version number that p has received. Processor q may not be the current owner, but this is correct, because, according to LRC, p does not necessarily need to see the latest write, but only the latest write by a processor with which it has synchronized. In either case, read fault or write fault, whole pages are sent, without any twinning or diffing.

A SW protocol uses memory only for owner write notices. Since the last owner's copy is always up-to-date, *garbage collection* of old owner write notices is done on-the-fly without extra synchronization between processors.

Our SW protocol improves on the original CVM protocol in the mechanism used for locating valid pages on *read* faults. In the CVM protocol, the faulting processor requests a page from the owner, possibly after forwarding through the home. Thus, a page request on a read miss can take 2 or 3 messages depending on whether the current owner happens to be the home or not. In our protocol, a faulting processor always asks for a page from the processor named in the owner write notice with the highest version number that it has received. Read faults are therefore always serviced in two messages. However, on *write faults*, the last version of the page needs to migrate to the new owner, thus the exact location of the last owner needs to be determined.

In our implementation of SW, as in the CVM protocol and Mirage [8], we address the ping-pong problem by guaranteeing a processor ownership for a newly obtained page for a minimum quantum of time before it can be taken away by another processor. We use a fixed 1 millisecond quantum. The results do not appear to be sensitive to the exact value of the quantum.

3 The Adaptive Protocols

Our adaptive protocols choose dynamically, on a per-page basis, whether to use a SW or a MW protocol. The choice is fully automated, and no user or compiler annotations are required. Instead, the run-time system monitors the shared memory access patterns, and decides on the appropriate protocol accordingly. Roughly speaking, pages in MW mode use the TreadMarks twinning and diffing protocol, while pages in SW mode use an extension of the CVM SW protocol. Pages may be in a transitional state where some processors have the page in SW mode and others have it in MW mode.

Our protocols adapt based on two different characteristics in an application's shared data access patterns: write-write false sharing and write granularity.

3.1 Adapting to Write-Write False Sharing

On each processor a state variable is associated with each page indicating whether the processor believes this page is in SW or MW mode. At times, this variable may have different values on different processors. When the state variable indicates that the page is in SW mode, the processor checks for the occurrence of write-write false sharing, and, if so, switches to MW mode (see Section 3.1.1). Conversely, when the state variable indicates that the page is in MW mode, the processor checks for the absence of write-write false sharing, and, if so, switches to SW mode (see Section 3.1.2).

3.1.1 Detecting Write-Write False Sharing in Single Writer Mode

Principle and Examples If there is no write-write false sharing on a page, then all writes to that page must be totally ordered by synchronization, or, in other words, by *happens-before-1*. A modification to the SW protocol for locating and transferring ownership allows this condition to be checked efficiently. The principle is: There is no write-write false sharing if and only if the processor taking a write fault and trying to get ownership knows the correct location of the owner and the correct version number for the page.

Before giving the details of the protocol, we illustrate the principle with a few examples.

- Consider the case where there is no write-write false sharing on a particular page. For instance, say processor p_1 acquires a lock, writes on the page, and then releases the lock. At the time a processor p_2 acquires the lock, it receives a write notice containing the version number of the page, and invalidates the page. When it then takes a write fault on the page and requests ownership, it knows the correct owner and version number of the page.

- Consider next the case where there is write-write false sharing. Continuing the above example, assume that, after processor p_2 writes to the page, p_1 writes to a different part of the same page without synchronizing with p_2. Processor p_2 does have the right version number at the time of its write, because there is no write-write false sharing at this point. It will become the new owner and increment the version number. When p_1 writes to the page, it no longer has an up-to-date value of the version number, indicating the onset of write-write false sharing.

- Finally, consider the case where processor p_1 acquires the lock l_a, writes on data item a in the page, releases the lock l_a, acquires the lock l_b, writes on data items b in the same page, and releases the lock l_b. Processor p_2 acquires l_a, writes on data item a, and becomes the owner of the page. Now assume that processor p_3 acquires lock l_b to write on data item b. When it writes on b, the resulting page fault causes it to request ownership from p_1, but p_1 is no longer the owner, signifying that write-write false sharing has occurred.

The Ownership Refusal Protocol At an acquire, the releasing processor creates write notices for each page that it has modified. For a page in SW mode, the owner creates an *owner write notice*, containing the processor id, the version number of the page, and the vector timestamp of its current interval. For a page in MW mode, a *non-owner write notice* is created, containing only the processor id and the current vector timestamp.

On a write fault to a page in SW mode, a processor tries to achieve ownership. In contrast to the SW protocol, there is no notion of a "home" to locate the owner. Instead, the processor uses the owner write notice with the highest version number, and sends the ownership request to the processor from which it has received that write notice (i.e., to the *last perceived owner* of the page). It includes in this message the version number in that write notice. If that processor is no longer the owner, or if the version number has changed, write-write false sharing has been detected and the ownership request is *refused*. Otherwise, ownership is *granted*. In either case, unlike with the SW protocol, ownership requests are never forwarded and always involve two messages.

If ownership is granted, the new owner increments the page's version number and makes the page writable. If the ownership request is refused, the requester puts the page in MW mode. It creates a twin and will later make a diff as in the MW protocol. If the target processor of the ownership request is still the owner, it maintains its ownership status until the next release. At that point, it generates an owner write notice for the page, but then drops ownership and puts the page in MW mode. Although at first glance it would seem appealing to drop ownership immediately at the time of the incoming ownership request, this is not possible because the owner does not have a twin, and therefore cannot make a diff.

The advantage of the adaptive protocol over the SW protocol is that it does not suffer from the ping-pong effect. The disadvantage compared to the MW protocol is the need for ownership messages. However, in the case of a write fault on an invalid page, the ownership request gets piggybacked on the page request, which was already present in the MW protocol.

Figure 1 demonstrates the behavior of the protocol with three different access patterns: producer-consumer, migratory, and write-write false sharing. In all three examples, processor p_1 is the initial owner, and has created an owner write notice with version number v_1. Processor p_2 synchronizes with p_1, and receives this write notice. As a result, v_1 becomes the highest version number p_2 knows about for this page (denoted HVN in the figure). In the producer-consumer pattern, p_2 takes a read fault on the page. This causes the page to move, but ownership stays with p_1. When p_1 later writes on the page again, it is still the owner and does not need to create a twin. In the migratory access pattern, the read fault by p_1 causes the page to move, and the subsequent write fault causes ownership to be migrated, so that p_2 can write on the page without making a twin. Finally, in the case of write-write false sharing, the write fault by p_2 also results in an ownership request message, but this request is refused by p_1, because p_1 has already written on the page and increased its version number to v_2 as a result. In this case, p_2 has to make a twin.

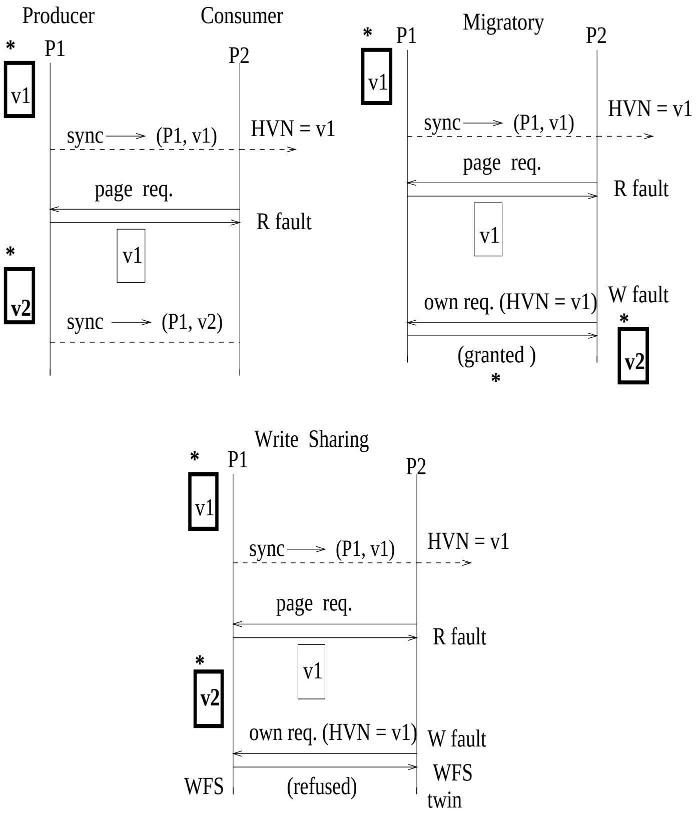

Figure 1: Behavior of the protocol with three access patterns: producer-consumer (top left) and migratory (top right), and write-write false sharing (bottom)

Merging Single Writer Copies and Diffs While there is a single writer for a page, processors receive only owner write notices. On an access miss, the whole page is requested from the last perceived owner. During the transition from SW to MW, owner copies and non-owner copies of a given page may co-exist.

Some method is needed for merging these two types of copies.

The merging of modifications is done by requesting the page from the *last perceived owner*, and applying the necessary diffs to that copy according to their timestamps. In more detail, when taking a fault on an invalid page, a processor looks at its list of write notices. The list contains write notices that indicate modifications of other processors and also local write notices. If the list contains only non-owner write notices, then the processor just needs to get the corresponding diffs from the other processors and apply them to its current copy. If the list contains one or more owner write notices, the processor selects the owner write notice with the highest version number, and obtains a copy of the page from the processor named in that write notice. It deletes all the write notices that are *dominated* by this owner write notice. The remaining write notices identify modifications to the page that happened either concurrent with or after the modifications reflected in the copy of the page that was just retrieved. The processor gets the diffs corresponding to these write notices, unless it has them already, and applies all the diffs to the page in timestamp order. The processor will not need to apply this special merging procedure again unless it sees a new owner write notice as a result of a switch back from MW to SW.

The global *garbage collection* of diffs is done at barriers as in the TreadMarks protocol (see Section 2.2). It differs in that only the last owner validates its copy by applying all the necessary diffs. Because garbage collection of diffs involves global synchronization as in TreadMarks, the last perceived owner is in fact the last owner of the page. On future access misses, all processors will thus retrieve the owner's copy of the page. In contrast, in TreadMarks, all concurrent writers of a page validate their copy. Furthermore, in the adaptive protocol, the garbage collection of old write notices can be done on-the-fly. Any write notice that becomes dominated by an owner write notice, including all old owner write notices, can be discarded.

3.1.2 Detecting the Absence of Write-Write False Sharing in MW Mode

When in MW mode, the adaptive protocol checks for the absence of write-write false sharing on a page. The principle here is: There is no write-write false sharing if there is a write notice for the page that dominates all other write notices. The adaptive protocol uses three extensions to the TreadMarks MW protocol to check for the absence of write-write false sharing.

First, processors piggyback information on diff requests indicating whether they perceive the page as write-write falsely shared or not according to the write notices they received. Each writer of a page monitors this false sharing information. Whenever a diff request comes in, the writer updates its local information to reflect the false sharing information received from the requester. Ownership requests to the *last perceived owner* are resumed if information collected from all processors in the approximate copyset for the page says that they see the page in SW mode. This approximate copyset is already maintained by the TreadMarks MW protocol. Second, as soon as a processor sees a new owner write notice and no concurrent secondary write notices, it infers that write-write false sharing has stopped. Third, at barriers all processors become up to date with all existing modifications. If at a barrier a processor receives a write notice for a page that dominates all other write notices, that processor can infer that write-write false sharing has stopped.

3.2 Adapting to Write Granularity

The underlying idea is that, even for pages for which there is no write-write false sharing, it might still be profitable to use diffing, if the size of the modifications to the page is small. The cost of twinning, diffing, and transferring a small diff may be cheaper than transferring a whole page. Besides the write granularity of the application, this tradeoff is highly dependent on the network bandwidth.

We use a simple threshold value to decide whether or not to use diffs. If the size of the modifications to a page is bigger than the threshold value, we switch to SW mode, otherwise we keep the page in MW mode. The threshold for a particular configuration is set at the value at which the cost of twinning, diffing, and transmitting the diff is equal to the cost of transmitting the entire page. While this threshold does not take into account other factors such as the increased memory usage and garbage collection overhead of MW, these factors are hard to quantify, and we found that the results are not very dependent on the exact value of the threshold.

Adapting to write granularity also alleviates the diff accumulation problem [17] that occurs in the MW protocol. Diff accumulation occurs in connection with migratory data where a sequence of synchronizing processors write the same data one after another. If a processor reads the data written by one of the writers, diffs from all of the preceding writers need to be applied, even if the modifications overwrite each other. This causes extra data to be sent. If the diffs are small, then several of them can be sent in a single message, limiting the resulting overhead. Diff accumulation becomes a serious problem, however, if the diffs are large. Our protocol addresses this problem by switching the pages with large diffs to SW mode.

3.3 Protocols Used in the Experiments

We use four protocols in the evaluation. The WFS protocol adapts to write-write false sharing in the manner described in Section 3.1. The WFS+WG protocol, in addition, adapts to the write granularity as described in Section 3.2. In both the WFS and WFS+WG protocols, all pages start in SW mode. The WFS+WG protocol, however, switches a page to MW mode as soon as the page becomes read-write or write-write shared. This enables the protocol to measure the write granularity. Afterwards, WFS+WG adapts to SW as described in Sections 3.1 or 3.2, with priority to the test for write-write false sharing. In other words, if the state variable indicates the presence of write-write false sharing, the page is placed in MW mode. If, however, the state variable indicates the absence of write sharing, the mode of the page is decided depending on the size of the diffs. As a baseline for comparison we include the MW and SW protocols.

4 Experimental Environment

Our experimental environment consists of 8 SPARC-20 model 61 workstations connected by a 155 Mbps ATM network. The processes communicate with each other over UDP sockets. The minimum round-trip time using send and receive for the smallest possible message is 1 millisecond. A remote access miss, to obtain a 4096 byte page from another processor, takes 1921 μseconds. A twin and full page diff take an average of 104 and 179 μseconds, respectively. To set the threshold for WFS+WG protocol, we measured the cost of twinning, diffing and sending the diff for different diff sizes. This led us to a conservative threshold value of 3KB to switch from MW to SW in WFS+WG.

5 Applications

We use 8 applications in this study: Red-Black SOR and TSP; Water and Barnes-Hut from the SPLASH benchmark suite [19]; IS and 3D-FFT from the NAS benchmark suite [4]; Shallow from NCAR [18] and ILINK, a large computational genetics code [11]. The applications and input sets vary considerably in terms of the amount of write-write false sharing and the write granularity.

Tables 1 and 2 summarize the relevant characteristics of the applications. Table 1 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. Table 2 provides the prevailing write granularity, and the percentage of shared pages that are write-write falsely shared. A large write granularity implies a size above our 3KB threshold. Variable means that the size of the writes vary with time. The write granularity and write-write false sharing data in the table are only valid for the particular input set used. Some applications (e.g., SOR, Water and Shallow) show variation in write granularity and write-write false sharing behavior depending on the input set.

Application	Data size	Sync	Time (sec.)
IS	21 x 14	l,b	7.8
3D-FFT	64x64x64	b	40.8
SOR	1000 x 2000	b	820.1
Water	512 molecules	l,b	48.7
TSP	19 cities	l	118.3
Shallow	1024 x 256	b	86.5
Barnes	32K bodies	l,b	242.0
ILINK	CLP 2x4x4x4	l,b	1388.3

Table 1: Applications, input data sets, synchronization (l=locks, b=barriers), and sequential execution time

Application	Write gran.	WS(%)
IS	large	0.0
3D-FFT	large	0.03
SOR	variable	0.0
Water	medium	3.5
TSP	small	1.0
Shallow	med-large	13.9
Barnes	small	61.9
ILINK	small	58.3

Table 2: Write granularity and the percentage of write-write falsely shared pages

6 Results

We first compare the speedups of the four protocols. We then present a detailed breakdown of the memory overheads and the communication requirements. Finally, we explain the results for each application. Unless otherwise noted, all results refer to 8-processor executions.

6.1 Execution Times

Figure 2 shows the speedup on 8 processors for each of the applications using the four protocols.

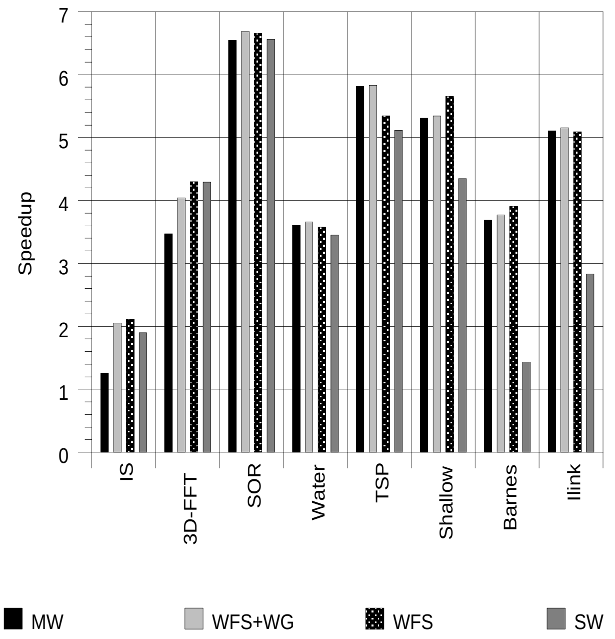

Figure 2: Speedup comparison on 8 processors: MW, WFS+WG, WFS, and SW

We first compare the non-adaptive SW and MW protocols. As expected, the amount of write-write false sharing determines the tradeoff. The SW protocol performs better than MW on applications with zero or very low write-write false sharing (IS and 3D-FFT), performs comparably on applications with low write-write false sharing (Water), and worse for applications with medium or high write-write false sharing (Shallow, Barnes, ILINK). Although there is no write-write false sharing in SOR, SW and MW perform comparably because of the high computation-to-communication ratio. For TSP, write-write false sharing is low, but the MW protocol performs better because of the small size of the diffs. The biggest relative speedup differences in favor of SW occur for IS and 3D-FFT where SW has a speedup of 1.9 and 4.3, respectively, vs. 1.2 and 3.5 for MW. The biggest differences in favor of MW occur for Barnes and ILINK where MW has a speedup of 3.7 and 5.1, respectively, compared to 1.4 and 2.8 for SW. These results are similar to those of Keleher [13], allowing for some differences in platform and applications.

Comparing the adaptive to the non-adaptive protocols, we see from Figure 2 that, for all but 3D-FFT and TSP, both adaptive protocols match or exceed the speedup of the best of the non-adaptive protocols. For these six applications, the speedups of WFS and WFS+WG are almost identical, with a slight edge for WFS with Shallow and Barnes, and a slight edge for WFS+WG with Water. For 3D-FFT, the WFS protocol matches the performance of SW, the best non-adaptive protocol. The WFS+WG protocol slightly lags SW, but performs better than MW. For TSP, the WFS+WG protocol matches the performance of MW, the best non-adaptive protocol. The WFS protocol lags MW, but beats SW.

Evaluated in terms of speedups, we conclude that our adaptation to write-write false sharing works well, with the adaptation to the write granularity having only a secondary effect (TSP) and occasionally having a small negative effect (3D-FFT).

A more subtle point is that *per-page* adaptation pays off. This is shown, for instance, by the improvements of WFS over MW on medium-high write sharing problems (Shallow, Barnes). Our WFS protocol performs better than the MW protocol on these problems because not *all* pages are write-write falsely shared. Similarly, WFS+WG has an edge over MW for most applications because, even if the average write granularity is small, most applications have large writes on some pages (see Section 6.4).

6.2 Memory Overhead

SW does not use twins or diffs. Total memory usage in the other three protocols is dominated by the memory used for twins and diffs. Table 3 presents the amount of twin and diff space used for all applications under these three protocols. Additional memory is used in all protocols for storing write notices, but this amount is small in comparison and not presented.

Prog	Protocol	Diffs (MB)	Twins (MB)
IS	MW	2.71	2.92
	WFS+WG	0.32	0.47
	WFS	0.00	0.00
FFT	MW	23.67	31.69
	WFS+WG	3.51	7.18
	WFS	0.01	0.18
SOR	MW	68.47	116.11
	WFS+WG	11.91	50.91
	WFS	0.00	0.00
Water	MW	3.55	13.09
	WFS+WG	0.62	4.10
	WFS	0.17	1.23
TSP	MW	0.33	15.05
	WFS+WG	0.21	13.64
	WFS	0.10	2.49
Shallow	MW	31.59	64.07
	WFS+WG	1.91	22.24
	WFS	0.71	16.43
Barnes	MW	21.08	105.00
	WFS+WG	17.51	101.73
	WFS	16.57	95.71
ILINK	MW	15.36	138.89
	WFS+WG	14.75	137.52
	WFS	10.57	118.49

Table 3: Memory consumption for MW, WFS+WG, and WFS

SW uses neither twins nor diffs, and so has the lowest memory overhead, followed by WFS. WFS substantially reduces the total memory consumption compared to MW. For applications that have no write-write false sharing (SOR and IS), the WFS protocol does not create any twins or diffs. In general, the amount of memory consumed in WFS is far lower than MW with the exception of the applications with high write-write false sharing (ILINK and Barnes). As a hybrid protocol, WFS+WG uses more memory than WFS for all applications, but has lower memory cost than MW.

Figure 3 uses 3D-FFT as an example of how our WFS+WG and WFS protocols adapt. The figure shows the total number of diffs on all processors as a function of time during the first 6 iterations. 3D-FFT overwrites entire pages during each iteration, and therefore most diffs are 4K in size. We can see that the diff space is rapidly consumed in MW, corresponding to the first steep ascending part of the graph. When the diff space exceeds the threshold (1MB) on one of the processors, garbage collection occurs at the next barrier. Each drop in the graph corresponds to a garbage collection. The WFS protocol uses diffs *only* when there is write-write false sharing. In 3D-FFT, only 0.03% of the pages exhibit write-write false sharing, so diff space used is negligible. The WFS+WG protocol starts out making diffs at the same rate as MW. However, as it sees that the diff size of each page is above the allowed threshold, it switches to a SW protocol for these pages. As pages are switched to SW mode the slope of the curve flattens. Finally, after the second iteration, WFS+WG does not create

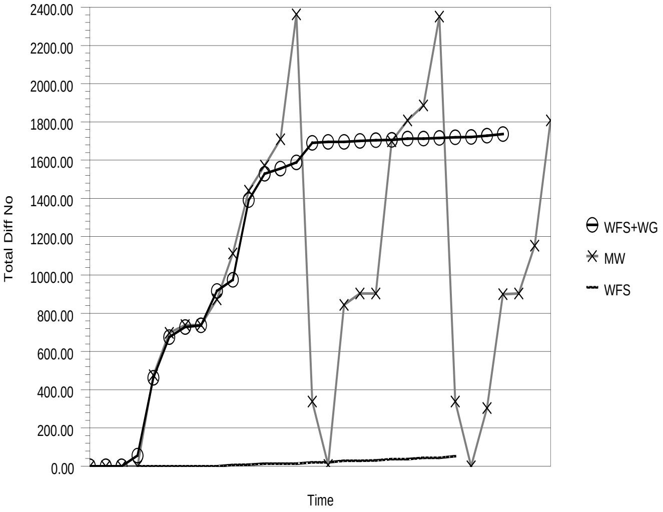

Figure 3: Diff creation and garbage collection patterns for the MW, WFS+WG and WFS in 3D-FFT

any more large diffs (corresponding to the flat area of the curve).

6.3 Communication

Table 4 provides data on the amount of communication for each of the 8 applications with each of the 4 protocols. Besides the number of messages and the amount of data, we also indicate the number of ownership requests for the WFS+WG, WFS and SW protocols. For ease of comparison, the number given represents ownership requests and not ownership related messages. For WFS and WFS+WG, the total number of ownership related messages is double the number of ownership requests. For SW, ownership requests may involve forwards, which are included in the total number of messages but not in the ownership request number.

The SW protocol sends the largest number of messages and the largest amount of data. For applications with high write-write false sharing, the difference between SW and the other protocols is due to the ping-pong effect. For applications with low write-write false sharing the slight increase is due to forwarding of ownership requests.

The tradeoff between MW and the adaptive protocols is not so uniform. For some applications, the ownership requests cause the adaptive protocols, especially WFS, to send more messages than MW. For Shallow, Barnes and 3D-FFT, the adaptive protocols, in particular WFS, send fewer messages than MW, because of the high number of messages exchanged during MW garbage collection.

6.4 Detailed Discussion

In the following discussion, we focus on the write-write false sharing and write granularity in each of the applications, and how our protocols adapt to these characteristics.

Program	Protocol	Msg (10^3)	Owner (10^3)	Data (MB.)
IS	MW	3.25	0.00	20.46
	WFS+WG	4.54	0.75	5.73
	WFS	4.77	0.75	5.32
	SW	5.10	0.75	5.44
3D-FFT	MW	21.52	0.00	42.59
	WFS+WG	18.79	0.00	37.23
	WFS	18.79	0.00	37.26
	SW	20.38	0.01	37.30
SOR	MW	87.77	0.00	83.16
	WFS+WG	89.38	0.87	84.85
	WFS	89.38	0.87	125.92
	SW	89.38	0.87	84.85
Water	MW	49.00	0.00	17.53
	WFS+WG	51.58	1.69	22.71
	WFS	52.87	2.30	24.99
	SW	54.70	2.64	36.30
TSP	MW	18.20	0.00	5.70
	WFS+WG	18.16	0.21	8.77
	WFS	21.44	2.18	29.54
	SW	25.44	2.89	31.90
Shallow	MW	39.11	0.00	51.03
	WFS+WG	45.72	3.41	53.38
	WFS	39.00	0.01	55.71
	SW	55.65	8.30	79.80
Barnes	MW	224.49	0.00	132.24
	WFS+WG	196.90	3.77	155.62
	WFS	196.84	3.81	156.86
	SW	831.83	274.90	1286.60
ILINK	MW	252.37	0.00	123.35
	WFS+WG	236.95	0.12	129.72
	WFS	236.98	0.35	229.86
	SW	313.44	28.73	479.78

Table 4: Number of messages, ownership requests, and amount of data exchanged for the four protocols

IS ranks an unsorted sequence of keys using bucket sort. The keys are divided among the processors. At first, processors count their keys in their private buckets. In the next phase, the values in the buckets are summed up. The sharing pattern in IS is migratory: the shared buckets are passed from one processor to another, protected by locks. There is no write-write false sharing, and the pages containing the shared buckets are completely overwritten by each processor. WFS keeps all these pages in SW mode during the entire execution. WFS+WG switches to SW mode for all pages after the first iteration. Although the adaptive protocols send more messages than MW due to ownership requests, diffing overhead and diff accumulation [17] in MW result in poorer performance relative to the adaptive protocols. The small improvement of WFS over SW is due to the extra messages in SW for forwarding ownership requests.

3D-FFT solves a differential equation using 3D forward and inverse FFT's. The phases are separated by barriers, with a transpose being performed to optimize the computation. Communication occurs in performing the transpose, and is of a producer-consumer nature. Write-write false sharing occurs on only one page out of 3072 total shared pages, and the modifications for this page are small (28 bytes). The other shared pages are completely overwritten almost every time they are touched. In WFS, each of the 8 processors switch once from SW to MW for the page for which there is write-write false sharing. WFS+WG switches to SW mode during the second iteration for all pages except for the write-write falsely shared page. Under the adaptive protocols, as in SW, access misses are handled by merely retrieving a copy of the page from another process, adding no additional diff creation/application overhead. MW creates diffs describing each modification because every page of data is replicated over the course of the execution. WFS and SW have identical performance, slightly better than WFS+WG because of the first iteration, and significantly better than MW.

In **SOR** the shared data structure is a matrix divided into roughly equal size bands of rows. Bands are assigned to processors. The program iterates over the matrix computing a new value for each element based on its four neighbors. Communication occurs across the boundary between bands. There is no write-write false sharing for the input size that we used. The boundary elements of the matrix are initialized to 1, and the internal elements are initialized to 0. In the early iterations, few elements change. However, the number of modifications grows with every iteration. WFS+WG starts out making diffs, because the write granularity is below the threshold in the early iterations. It switches to SW mode after the first 86 iterations. WFS+WG has the best performance of all protocols. Since the computation-to-communication ratio in this application is very high, all protocols perform well.

Water is a molecular dynamics simulation. It computes the intra- and inter-molecular forces using an $O(n^2)$ algorithm with a cut-off radius. The main shared data structure is the array of molecules allocated contiguously and partitioned among processors. Depending on how array partitions align to pages there may or may not be false sharing. On average, 6 molecule data-structures are allocated to the same page, and write-write false sharing occurs on 3.5% of the pages. In WFS, 2-3 pages (from a total of 85 shared pages) switch to MW. WFS+WG switches to moving whole pages for 18% of the pages. However, the size of writes on these pages is only slightly above the 3K threshold leading to marginal improvement over MW.

TSP uses a branch-and-bound algorithm to find the minimum tour that starts at a designated city, passes through every other city exactly once, and returns to the original city. In TSP, write granularity is small. For example, an update to the shared tour queue modifies a couple of words. There is little write-write false sharing in TSP. WFS switches from SW to MW on a total of 2 pages for all 8 processors (out of 196 shared pages) and thus moves whole pages for the most part. WFS+WG uses mostly diffs because only for one page it observes a large diff (> 3K) which causes it to put the page in SW mode. The increase in the amount of data and the ownership messages cause WFS and SW to perform worse than WFS+WG and MW.

Shallow solves difference equations on a 2D grid for weather prediction. Parallelization is done in bands, with sharing only across the edges. In our 1024×256 element input set, write-write false sharing occurs on 13.9% of the pages. Shallow makes a clear case for per-page adaptation. The WFS protocol performs better than both non-adaptive protocols. The WFS protocol switches to MW mode for all of the write-write falsely shared pages, and keeps the other pages in SW mode. The WFS protocol outperforms the SW protocol because of the ping-pong effect, and the MW protocol because of lower diffing and twinning overheads. In addition, the WFS protocol uses fewer messages than the MW protocol. The two reasons are that almost all ownership messages can be piggy-backed on page requests, and the WFS protocol's garbage collection policy only validates one copy of the page. Also, memory consumption in the WFS protocol is much lower. It creates 75% fewer diffs and uses 98% less memory than the MW protocol. The WFS+WG protocol performs identically to the MW protocol and outperforms the SW protocol. Even though most of the pages have a single writer, the small size of the diffs keeps the pages in the MW mode. Only 7.2% of the pages change to SW mode.

Barnes-Hut simulates the evolution of a system of bodies under the influence of gravitational forces. It uses a hierarchical tree-based method to compute forces between bodies. The space is broken into cells. The internal nodes of the tree represent the cells, and the leaves represent the bodies in the corresponding cells. The tree is built in each time step by loading bodies into it. The bodies are partitioned among processors, and then each processor computes the forces on its bodies and updates the positions and velocities of the bodies. In the version that we use, the array of bodies is shared, and the cells are private. Both read and write accesses to the global body array are fine grained. All pages containing this array (61.9% of the shared pages) are write-write falsely shared. The SW protocol performs poorly. It sends more messages, and moves more data than the other protocols. The adaptive protocols slightly outperform the MW protocol. They switch to the MW mode for all of the pages containing bodies, while keeping the other pages in SW mode. Even though the adaptive protocols move more data, they use fewer messages and generate fewer twins and diffs than the MW protocol.

ILINK is a genetic linkage analysis program that locates specific disease genes. The main data structure is a pool of sparse arrays called `genarrays`. A master processor assigns the nonzero elements to all processors in a round-robin fashion. After each processor has worked on its share of non-zero values, the master processor sums up the contributions. The pool of sparse genarrays is in shared memory and all processors access it. Thus, the dominant access pattern is write-write false sharing (58% of the pages) and WFS adapts to MW mode for these pages. There is a slight decrease in the total message count for the adaptive protocols compared to MW, as a positive side effect of the garbage collection method used. However, the WFS protocol moves more data for the non write-shared pages, due to the sparse data structures used in the application and this shows in the total data movement. WFS+WG adapts to SW mode for an average of 1-2 pages per processor, the only pages for which diffs are large, and thus gets a slight improvement compared to MW. This program has a lot of write-write false sharing, but the computation to communication ratio is higher than Barnes', and thus all protocols have better speedup.

6.5 Summary

The results confirm the benefits of adaptation to write-write false sharing: avoiding the ping-pong effect if write-write false sharing is present, and avoiding the costs of twinning and diffing in its absence. Adaptation to write granularity has only a second-order effect for this set of applications and this environment, and introduces a hysteresis in the protocol that can lead to performance degradation and extra memory consumption. The results also demonstrate the benefit of per-page adaptation: The adaptive protocols sometimes outperform both non-adaptive protocols. A side effect of the adaptive and SW protocols is reduced memory utilization compared to the MW protocol. On the other hand, the ownership request messages in the adaptive protocols may cause some increase in the total number of messages.

7 Related Work

The SW protocol we use is based on the work of Keleher [13]. His work demonstrates that the performance benefits resulting from using LRC rather than sequential consistency (SC) are considerably larger than those resulting from allowing multiple writers. We show that under LRC the benefits of MW and SW protocols can be combined into one adaptive algorithm that uses the appropriate protocol on a per-page basis.

Various techniques have been proposed to replace diffing by cheaper alternatives [15, 22] or to offload diffing to a communication coprocessor [5, 23]. This work is orthogonal to ours, in that we could incorporate these techniques into our adaptive protocols. Using per-word timestamps [1, 15, 22] addresses the problem of diff accumulation directly. The problem is alleviated in our system because we switch to using whole pages whenever the diffs are large.

Cox and Fowler, and Stenstrom and Brorsson [7, 20] describe hardware cache-coherence protocols that adapt to migratory sharing patterns. Migratory cache blocks are detected automatically. If a processor first reads and then writes a block, these protocols invalidate the old copy and migrate ownership of the block to the new processor on the read miss rather than on the write hit. This strategy requires only one bus transaction, where otherwise two would be required: one to replicate the block on the read miss, and one to invalidate the old copy on the write hit. Our adaptive protocols could be extended to automatically detect migratory data access and optimize the ownership acquisition accordingly.

Munin [6] uses multiple protocols to handle data with different access characteristics. The innovation in our work is that it chooses automatically between SW and MW protocols. In Munin, the choice of protocol is based on somewhat burdensome user annotations.

MGS [21], a DSM system for distributed SMPs, uses a base protocol similar to Munin. Their protocol employs a single writer optimization that avoids diffing overhead when there is only one writable copy. Although the twin is still made, the entire page is sent to the home instead of computing a diff. The work of Zhou et. al [23] also avoids diffing when the home node is in fact the single writer for the page. In contrast, our adaptive protocols avoid twinning and diffing overhead without using a fixed home node. This avoids unnecessary message traffic if the home node is poorly chosen.

False sharing has also been addressed by compile-time analysis [12], remapping of data within the address space [9], and by using objects as a smaller consistency unit [22]. All of these techniques seek to eliminate rather than tolerate false sharing. They are, however, limited in their applicability due to either the requirement of a special-purpose language/compiler or restrictions on the applications.

8 Conclusions

We have introduced adaptive protocols for software DSM that dynamically choose between SW and MW mode on a per-page basis. The choice is based on the presence of write-write false sharing and/or write granularity. Write-write false sharing is detected by a new ownership refusal protocol. Our adaptive protocols do not require any user, compiler, or hardware support.

The adaptive protocols perform well, matching or exceeding the performance of the best of the non-adaptive approaches on 7 out of 8 applications. Adaptation to write-write false sharing is the most important factor contributing to performance. Adaptation to write granularity is a secondary factor in this environment and for this set of applications.

In the adaptive protocols, we avoid the worst case of both the MW protocol (the diff accumulation problem) and the SW protocol (the ping-pong effect). Furthermore, communication overheads are much lower than for the SW protocol, and memory consumption is significantly reduced compared to the MW protocol.

References

[1] S.V. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proceedings of the Second High Performance Computer Architecture Symposium*, pages 26–37, February 1996.

[2] S.V. Adve and M.D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[3] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[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] R. Bianchini, L.I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding communication latency and coherence overhead in software dsms. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 198–209, October 1996.

[6] 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.

[7] 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.

[8] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[9] V.W. Freeh and G.R. Andrews. Dynamically controlling false sharing in distributed shared memory. In *Proceedings of the Fifth Symposium on High-Performance Distributed Computing*, 1996.

[10] 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.

[11] S.K. Gupta, A.A. Schäffer, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Integrating parallelization strategies for linkage analysis. *Computers and Biomedical Research*, 28:116–139, June 1995.

[12] T.E. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the 5th ACM Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[13] 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.

[14] 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.

[15] P.T. Koch, R.J. Fowler, and E. Jul. Write ranges: A technique for improving capture and propagation of writes in software DSMs. Submitted for publication.

[16] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[17] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings SuperComputing '95*, December 1995.

[18] R. Sadourny. The dynamics of finite-difference models of the shallow-water equations. *Journal of Atmospheric Sciences*, 32(4), April 1975.

[19] 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.

[20] 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.

[21] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A multigrain shared memory system. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, May 1996.

[22] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.

[23] 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, November 1996.