Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory

Cristiana Amza[†], Alan Cox[†], Karthick Rajamani[‡], and Willy Zwaenepoel[†]
[†] Department of Computer Science
[‡] Department of Electrical and Computer Engineering
Rice University
{amza, alc, karthick, willy}@cs.rice.edu

Abstract

Software Distributed Shared Memory (DSM) systems based on virtual memory techniques traditionally use the hardware page as the consistency unit. The large size of the hardware page is considered to be a performance bottleneck because of the implied false sharing overheads. Instead, we show that in the presence of a relaxed consistency model and a multiple writer protocol, a large consistency unit is generally not detrimental to performance. We study the tradeoffs between false sharing and aggregation effects when using large consistency units. In this context, this paper makes three separate contributions:

1. We document the cost of false sharing in terms of extra messages and extra data being communicated. We find that, for the applications considered, when the virtual memory page is used as the consistency unit, the number of extra messages is small, while the amount of extra data can be substantial.
2. We evaluate the performance when the consistency unit is increased to a multiple of the virtual memory page size. For most applications and data sets, the performance improves, except when the false sharing effects include extra messages or a large amount of extra data.
3. We present a new algorithm for dynamically aggregating pages. In our algorithm, the aggregated pages do not necessarily need to be contiguous. In all cases, the performance of our dynamic aggregation algorithm is similar to that achieved with the best static page size.

These results were obtained by measuring the performance of eight applications on the TreadMarks distributed shared memory system. The hardware platform used is a network of 166Mhz Pentiums connected by a switched 100Mbps Ethernet network.

1 Introduction

Since Li and Hudak's seminal work [14] on software distributed shared memory (DSM) in 1985, the "Battle Against False Sharing" has been a dominant, if not the dominant, theme of research in this area. Today, it is generally accepted that the ill effects of false sharing can be reduced, but not entirely eliminated, using a relaxed memory consistency model and a multiple writer protocol [1, 3, 9, 13]. Despite this, the conventional wisdom remains that the overhead of false sharing, as well as fine-grained true sharing, in page-based consistency protocols is the primary factor limiting the performance of software DSM.

In a software DSM that uses lazy release consistency (LRC) [13] and multiple writer protocols [3], such as TreadMarks [1], false sharing can have two harmful effects: it can cause extra messages to be sent, and it can cause additional data to be sent on messages that also carry truly shared data. We refer to these as *useless messages* and *useless data*. For the applications in our test suite, when the virtual memory page is used as the consistency unit, the number of useless messages is small, while the amount of useless data can be substantial. The reason is that for most pages for which there is false sharing, there is also true sharing on the page, making the message necessary, but any extra data useless.

Next, we consider increasing the consistency unit to a multiple of the virtual memory page size. Increasing the consistency unit size has both positive and negative effects. It reduces the number of page faults and consistency operations, and the number of messages because of aggregation. It may, however, increase false sharing, and in turn increase the number of messages and the amount of data exchanged. For the majority of the applications and the data sets considered in this paper, performance gets better or stays the same with larger consistency units. Only when false sharing induces extra useless messages or when there is a large amount of useless data does performance degrade.

Motivated by the generally positive effect of increasing the size of the consistency unit, we present an algorithm that dynamically aggregates pages into larger *page groups*. On a page fault, all pages in a page group are validated at the same time. The pages in page groups need not necessarily be contiguous. The algorithm monitors the page faulting behavior of the individual pages, and decides whether to aggregate pages into page groups or whether to split page groups into pages. Dynamic aggregation avoids an increase in false sharing, while still aggregating pages into larger units. It does, however, incur a cost for monitoring the page faulting behavior. We found this cost to be small relative to the gains achieved. For all applications and data sets, the dynamic aggregation algorithm performs nearly as well as the best static page size.

Our application suite consists of eight programs. Barnes and Water are from the SPLASH benchmark suite [17]; 3D-FFT is from the NAS benchmark suite [2]; Ilink is a widely used genetic linkage analysis program [5]; Shallow is a benchmark developed at NCAR [16]; and Modified Gramm-Schmidt (MGS), Jacobi, and Traveling Salesman Problem (TSP) are simple computational kernels. Our platform is a 100Mbps switched Ethernet connecting 8 166Mhz Pentium machines. We use the TreadMarks software DSM [1] as the basis for our experiments. TreadMarks implements a lazy release consistent memory model [13]. It uses virtual memory page faults to detect access misses, and twinning and diffing to record modifications to a page and to implement a multiple writer protocol [3].

The rest of this paper is organized as follows. Section 2 discusses the effects of false sharing when the consistency unit is a virtual memory page. Section 3 describes the tradeoff between false sharing and aggregation when the consistency unit is statically increased to a multiple of the page size. Section 4 presents the dynamic aggregation algorithm. Section 5 discusses our performance measurements. Section 6 discusses related work. Section 7 provides our conclusions.

2 False Sharing Effects

The discussion of false sharing in this paper concerns page-based software DSM systems that use lazy release consistency (LRC) and multiple writer protocols, such as TreadMarks. With release consistency [9, 13], the modifications of processor p become visible to processor q only after q synchronizes with p. If an invalidate protocol is used, modifications cause a page to be invalidated after the synchronization. Access to an invalidated page causes a page fault, which in turn causes page fault request messages to be sent out to all of the processors that wrote the page concurrently before the synchronization.

In multiple writer protocols [3], write detection is done by twinning, and diffing. On the first write to a shared page, an identical copy of the page (a twin) is made. The twin is then compared with the modified copy of the page to generate a diff, a record of modifications to the page. This diff is returned in response to a page fault request.

Lazy release consistency and multiple writer protocols alleviate the worst effects of false sharing. In particular, concurrent reads and writes to the same page do not cause communication at the time of the access. False sharing may, however, still cause extra communication, either in the form of *useless messages* or in the form of *useless data* carried on messages that also carry useful data.

Useless messages are produced by write-write false sharing. For instance, assume that processors p_1 and p_2 write to the same page, p_1 to the top half and p_2 to the bottom half. After all processors come to a barrier, processor p_3 reads the top half of the page. Logically, processor p_3 only needs the data produced by p_1, and therefore a single message exchange with p_1 would suffice. The barrier, however, causes p_3's copy of the page to be invalidated by both p_1 and p_2. When p_3 reads the page and incurs an access miss, it must therefore request diffs from both p_1 and p_2. The messages exchanged with p_2 are *useless messages* caused by write-write false sharing.

The *useless data* occurs in conjunction with true sharing. However, we classify this effect as false sharing, because it implies communication not strictly required by true sharing. For instance, assume processor p_1 modifies the entire page, and then synchronizes with processor p_2. Assume that p_2 wants to read only the top half of the page. The synchronization causes p_2's copy of the page to be invalidated, in turn causing p_2 to take an access fault and request a diff from p_1. This diff contains the entire page, although p_2 will only read the top half. The bottom half of the page sent in the diff is *useless data.*

Previous research [15] has studied the extra communication in software DSM induced by false sharing. We argue that the study of false sharing, in isolation, is not a very good indicator of the effect of false sharing on program performance. Instead, we argue that the key to understanding the effect of false sharing is its relationship to true sharing. If, between two processors, both true and false sharing occur on the same page, then false sharing only causes an increase in data, in the form of useless data. Only when there is no true sharing between the two processors does false sharing cause extra messages.

In current networks of workstations, the cost of sending extra messages is typically much higher than the cost of sending some additional data on a message. The effect of false sharing is therefore primarily determined by the number of useless messages, and only to a second order by the amount of useless data. In Section 5 we illustrate the effects of false sharing on the applications in our test suite by breaking up the total number of messages and the total amount of data in useful and useless messages and data. As will be seen, two patterns emerge. For one class of applications, there is little false sharing, indicated by low values for both useless messages and useless data. For the other class, there are few useless messages, but the amount of useless data is substantial. This pattern indicates that there is false sharing, but that it occurs mostly on pages on which there is also true sharing.

For the latter class of applications, it appears promising to increase the consistency unit. Since most pages are truly shared (in addition to possibly being falsely shared), aggregation should be beneficial, with little additional false sharing being introduced. For the former class of applications, the tradeoff is less clear, given the potential for introducing false sharing where there previously was none. This tradeoff is explored in more detail in the next section.

3 Static Aggregation

Choosing a larger consistency unit has a number of advantages unrelated to communication such as reducing the number of access faults and the number of consistency operations including memory protection operations, twinning and diffing. Disadvantages include the need to twin and diff larger consistency units.

In terms of communication, larger consistency units form a tradeoff between aggregation and potentially increased false sharing. As with the discussion of false sharing with a particular size of consistency unit, this tradeoff needs to be understood in connection with true sharing. For instance, assume processor p_1 writes to two contiguous pages, then synchronizes with p_2, after which p_2 reads both pages. With pages as the consistency unit, this causes two message exchanges. If the consistency unit is doubled, only a single message exchange is necessary, and the amount of data exchanged remains the same. Some of the data in this message may be useless, but the amount of useless data remains the same as before. If we change the example slightly so that p_2 only reads the first page after synchronization, then the number of messages remains constant at one, but the amount of data exchanged increases when the consistency unit is doubled. When the consistency unit is a page, the modified data in the second page is not communicated, but when the consistency unit is twice the page size, it traverses the network (and becomes useless data). As a second example, consider the case where processor p_1 writes the first page, processor p_2 writes the second page, p_1, p_2, and p_3 synchronize, and finally p_3 reads both pages. Here, the number of message exchanges and the amount of data remain the same. There is still an advantage to the larger consistency unit, because p_3 can request both diffs in parallel, whereas with the page as the consistency unit, they would be requested in sequence. If we modify this second example, so that p_3 reads only the first page, there is only one message exchange, between p_1 and p_3, when the consistency unit is a page. When the consistency unit is doubled, there is an additional and useless message exchange between p_2 and p_3.

In general, the number of message exchanges occurring in response to a page fault is equal to the number of concurrent writers seen by the faulting processor at the previous synchronization, or

$$messages = access(P) \times card(CW(P))$$

in which $access(P)$ is 1 if the page is accessed and 0 otherwise, and $card(CW(P))$ is the cardinality of the set $CW(P)$ of concurrent writers to the page P before synchronization. When two contiguous pages P_a and P_b are aggregated into a single consistency unit (P_a, P_b), the number of message exchanges occurring in response to a page fault on the new consistency unit is

$$messages = access(P_a, P_b) \times card(CW(P_a) \cup CW(P_b))$$

The potential increase or decrease in the number of messages in going from the smaller to the larger consistency unit is then

$$access(P_a) \times card(CW(P_a)) + access(P_b) \times card(CW(P_b))$$
$$-access(P_a, P_b) \times card(CW(P_a) \cup CW(P_b))$$

This formula can be generalized to any larger consistency unit, and a similar formula can be derived for the amount of data exchanged.

Clearly, the evolution of the number of concurrent writers to the consistency unit is the critical performance factor. If there is a drastic increase in the number of concurrent writers to the larger consistency unit, then false sharing dominates, the number of useless messages increases, and performance deteriorates. Conversely, if the number of concurrent writers stays constant or increases slightly, then the benefits of aggregation result in an improvement of overall performance.

In order to capture this behavior, we characterize our applications by a *false sharing signature*, a histogram denoting the distribution of the number of concurrent writers (and therefore the number of message exchanges) observed at a page fault. A sizable shift in false sharing signature towards larger numbers when going to larger consistency units predicts a loss in performance. Conversely, if the false sharing signature largely remains invariant, performance increases with increasing size of consistency unit.

4 Dynamic Aggregation

Clearly, some applications may benefit from a larger consistency unit, while for others performance may deteriorate. As a solution, we present a dynamic aggregation algorithm that coalesces pages into *page groups*, based on their access patterns. 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* the pages of the group are requested at the first fault on any page that is a member of the group. Multiple requests addressed to the same processor are combined, resulting in fewer messages for the data transfer for the group as a whole. Even if the diffs need to 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.

Page groups are computed at each synchronization. Essentially, the algorithm groups pages that were accessed prior to the synchronization (up to some implementation-dependent maximum number of pages per group), so that it can request them as a group should they be accessed again after the synchronization.

In order to track the access pattern for a page, the algorithm keeps each page invalid until the first access to that page. The occurrence of a page fault signifies an access to the page. Note that a page may be kept invalid, even though it may already have been updated on the first access to another page of the group. If there is a fault for such a page in the group, then no data is requested because all updates have already been received. This strategy allows the algorithm to detect any change in the program's access patterns over the course of the execution.

Compared to the static use of a larger consistency unit, this simple dynamic algorithm captures the "good" cases of aggregation and avoids the "bad" cases of false sharing (see Section 3). When multiple pages are accessed by a processor, the algorithm records this fact, and fetches them together after the next synchronization, irrespective of the number of concurrent writers. This is exactly the behavior of static aggregation, when it is successful. Conversely, when the processor does not (or does no longer) access several pages together, the algorithm observes this behavior as well, and reverts to using pages, thereby avoiding the scenarios where false sharing has a detrimental effect on performance with static aggregation. In addition, the dynamic aggregation algorithm benefits from not being restricted to aggregating only contiguous pages.

The disadvantage of dynamic aggregation compared to the static use of large pages is the extra faults and memory protection operations necessary to keep track of the access patterns. Furthermore, there is a hysteresis effect in detecting changes to the access patterns which may cause some useless messages while the scheme reconstitutes the page groups. However, these useless messages are overlapped with the diff requests for the faulting page. Overall, these drawback are minor compared to the gains achieved by avoiding false sharing, as will be demonstrated by the performance results in Section 5.

5 Performance Results

5.1 Platform

Our experimental platform is a network of eight 166MHz Pentiums running FreeBSD 2.1.6. Each machine has 512 Kbytes of cache and 64 Mbytes of memory. The hardware page size is 4 Kbytes. The network connecting the machines

Program	Input Size	Seq. Time	Speedup
Barnes	16K	69.8	4.25
ILINK	CLP 2x4x4x4	1127.9	5.54
TSP	19	72.8	6.40
Water	512	75.0	4.43
Jacobi	1Kx1K	69.7	7.04
	2Kx2K	277.7	7.35
3D-FFT	64x64x32	18.7	4.07
	64x64x64	38.2	4.31
	128x128x128	–	–
MGS	1Kx1K	120.9	5.64
	2Kx2K	1112.4	6.51
	1Kx4K	560.3	6.11
Shallow	1Kx0.5K	179.1	5.01
	2Kx0.5K	–	–
	4Kx0.5K	–	–

Table 1: Applications, data sets, sequential execution times (in seconds), and speedups. No sequential execution times and speedups are presented when the problem size exceeded the available memory for one processor and paging was observed for the sequential run.

is a 100Mbps switched 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 296 microseconds on this platform. The time to acquire a lock varies from 374 to 574 microseconds. The time for an eight processor barrier is 861 microseconds. The time to obtain a diff varies from 579 to 1,746 microseconds.

5.2 Applications

We used eight applications in this study: Barnes and Water [17], 3D-FFT [2], Ilink [5], Shallow [16], Modified Gramm-Schmidt (MGS), Jacobi, and Traveling Salesman Problem (TSP).

Table 1 includes for each application, the data set sizes used, the sequential execution time, and the speedup on 8 processors using the hardware page size as the consistency unit. No sequential execution times and speedups are presented when the problem size exceeded the available memory for one processor and paging was observed for the sequential run. The applications and data set sizes were chosen to display a large variation in the amount and the effect of false sharing, as well as a large variation in the granularity with which shared data is accessed. For Barnes, Ilink, TSP and Water, the false sharing behavior is largely independent of the problem size, and thus we show results with only one input size. For Jacobi, 3D-FFT, MGS and Shallow, false sharing behavior varies with the problem size, and we present all the relevant cases. A detailed discussion of the applications is deferred until Section 5.5.

5.3 Measurement Results

Figure 1 shows the execution times, the number of messages, and the amount of data for Barnes, Ilink, TSP, and Water, with consistency units of 4, 8, and 16 Kbytes, and with the dynamic aggregation algorithm. All results are normalized to those with the virtual memory page size of 4 Kbytes. The messages and the data are broken down into useful and useless messages, and useful data and useless data. Useless data is in turn broken down into useless data carried in useless messages, and useless data carried in useful messages, which we refer to as piggybacked useless data. Figure 2 shows the same results for Jacobi, 3D-FFT, MGS, and Shallow. Figure 3 shows the evolution of the false sharing signature for Barnes, Ilink, Water, and MGS for 4 and 16 Kbytes. Each bar is broken down to reflect the number of useful and useless messages corresponding to that entry in the histogram.

The breakdowns of messages and data are obtained by instrumenting the programs. The instrumentation records all reads (loads) and writes (stores), and the application of all diffs. After applying a diff to a region of a page, if a word from that region is read before being overwritten, that word is counted as useful data. If a word is never read or overwritten before being read, it is counted as useless data. A useless message is a message that carries no useful data. Otherwise, a message is characterized as useful.

5.4 Overall Discussion

The results for Barnes, Ilink, TSP, and Water are similar. Performance improves with increasing consistency unit size. The execution time decreases, the number of messages decreases, and the amount of data either stays constant (for Ilink and TSP) or shows a very slight increase (for Barnes and Water). The evolution of the false sharing signature is also similar for these applications. Barnes, Ilink and Water are shown as examples in Figure 3. For Barnes and Water, there is a very slight shift to the right going from 4 to 16 Kbytes. For Ilink, there is virtually no change.

At 4 Kbytes, all of these applications have a relatively small number of useless messages, but a substantial amount of useless data. This indicates false sharing, but in such a way that false sharing is usually mixed with true sharing on the same page. It may seem paradoxical that increasing the consistency unit size improves performance for applications that already incur a substantial amount of false sharing at the smallest consistency unit size. The paradox is explained by the fact that in these applications, each processor makes fine-grained accesses to a large shared memory region. Thus, by increasing the consistency unit size, the aggregation effect plays its role, reducing the number of messages. False sharing does not increase, because each processor already accessed most of the shared memory region with the smallest consistency units. Thus, no extra useless messages and useless data are introduced by going to a larger size.

For Jacobi, 3D-FFT, MGS, and Shallow, the results are highly dependent on the problem sizes. For the smallest problem size, Jacobi, 3D-FFT, MGS, and Shallow show worse performance when going from 4 to 8 and 16 Kbyte consistency unit sizes. However, for the medium input sizes, 3D-FFT 64x64x64, MGS 2Kx2K, and Shallow 2Kx0.5K show improvement when going from 4 Kbytes to 8 Kbytes, but deteriorate when going to 16 Kbytes. Finally, Jacobi 2Kx2K, 3D-FFT 128x128x128, MGS 1Kx4K, and Shallow 4Kx0.5K improve when going to larger consistency units.

The only dramatic performance deterioration occurs with MGS, because of a very large increase in the number of useless messages. This behavior is also indicated by the sizable shift to the right in the false sharing signature for MGS, as shown in Figure 3. For the smaller data set sizes for Jacobi and 3D-FFT, the performance deterioration is due to an increase in the amount of useless data. For the smaller data set in Shallow, it is due to a combination of extra useless messages and useless data.

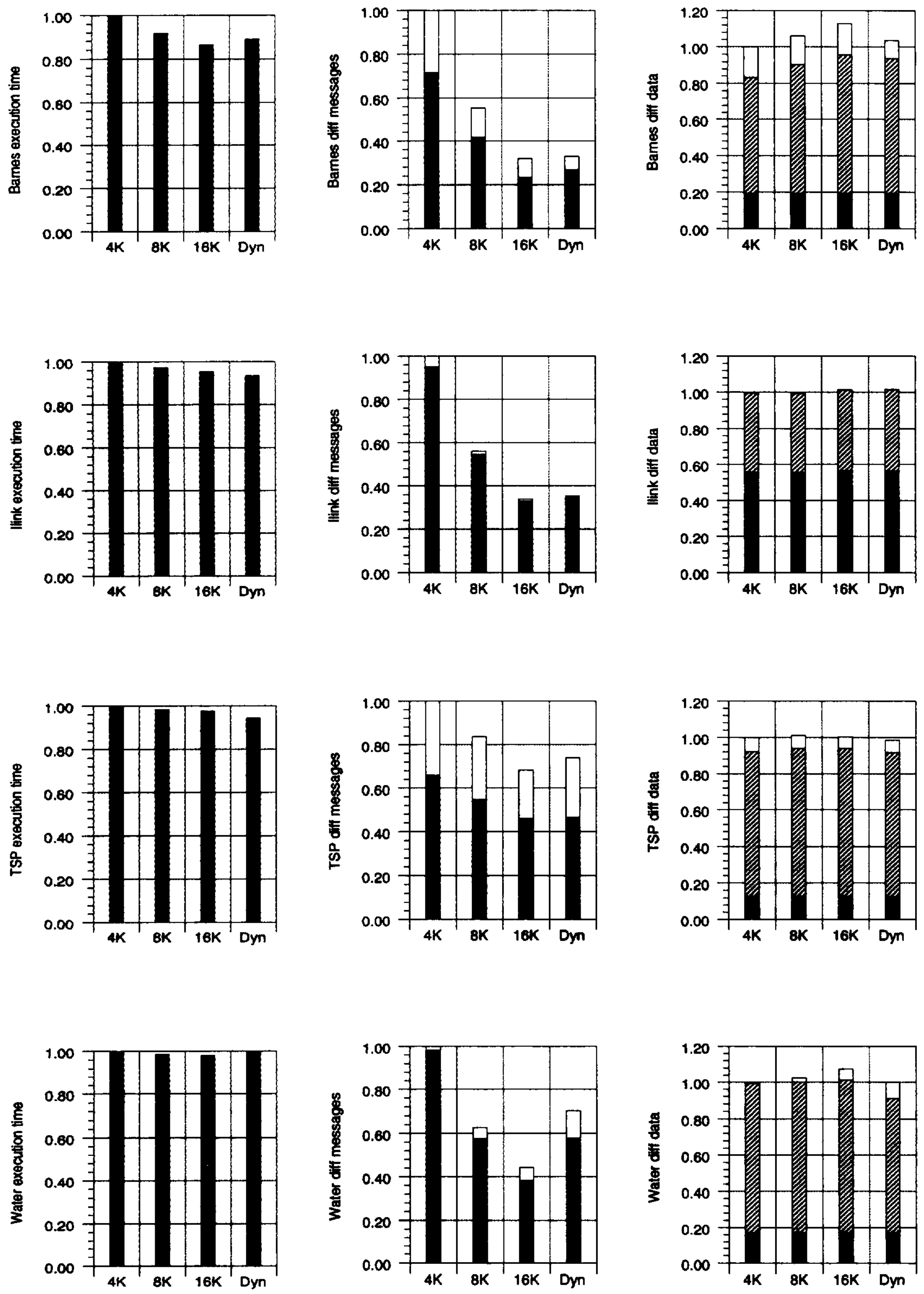

Figure 1: 8-processor execution times, messages, and data, normalized to the corresponding number for 4 Kbytes. Communication breakdown: black - useful, white - useless, striped - piggybacked useless data

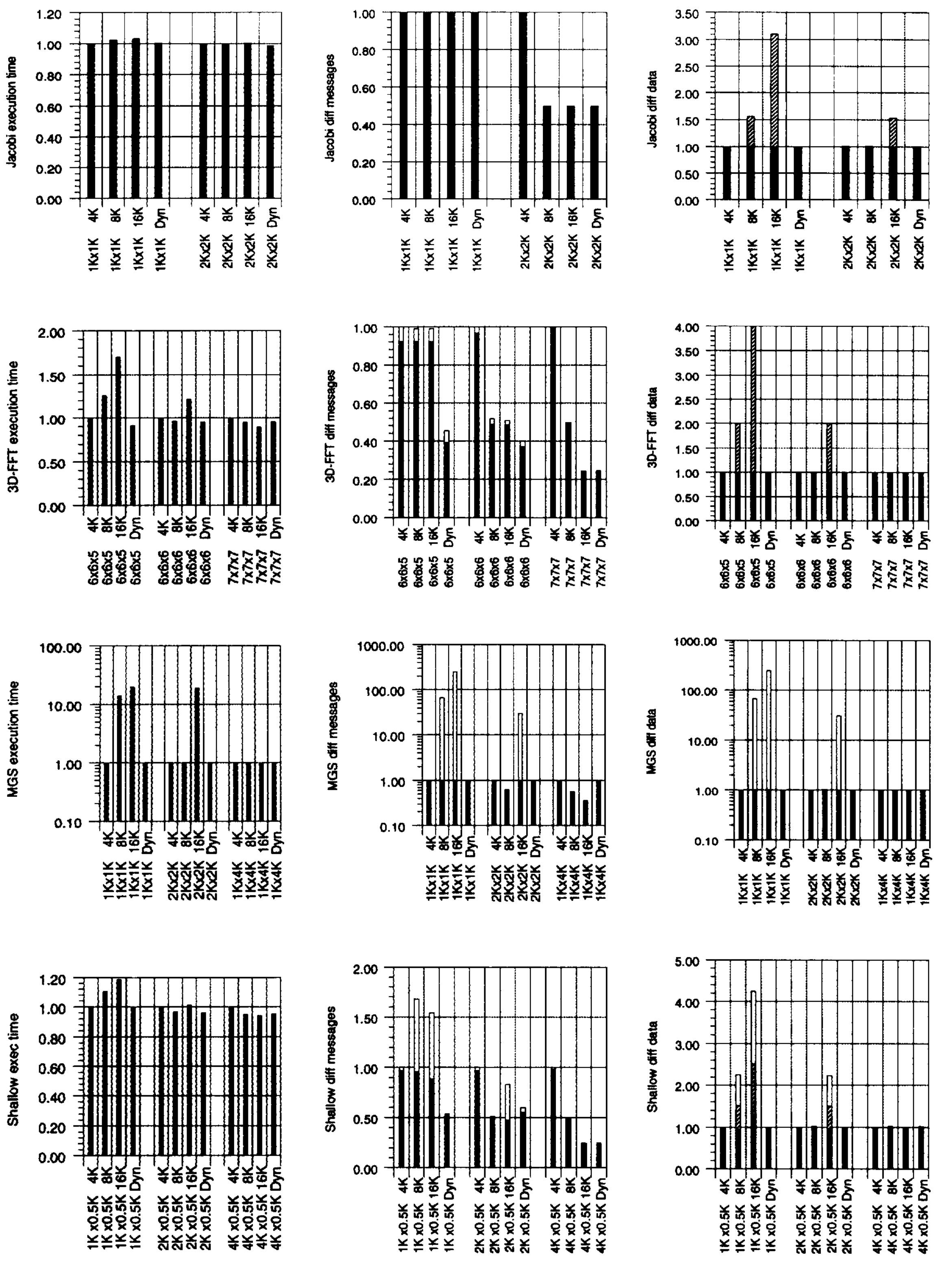

Figure 2: 8-processor execution times, messages, and data, normalized to the corresponding number for 4 Kbytes. Due to the severe degradation, all the statistics in MGS are represented on a logarithmic scale. Communication breakdown: black - useful, white - useless, striped - piggybacked useless data

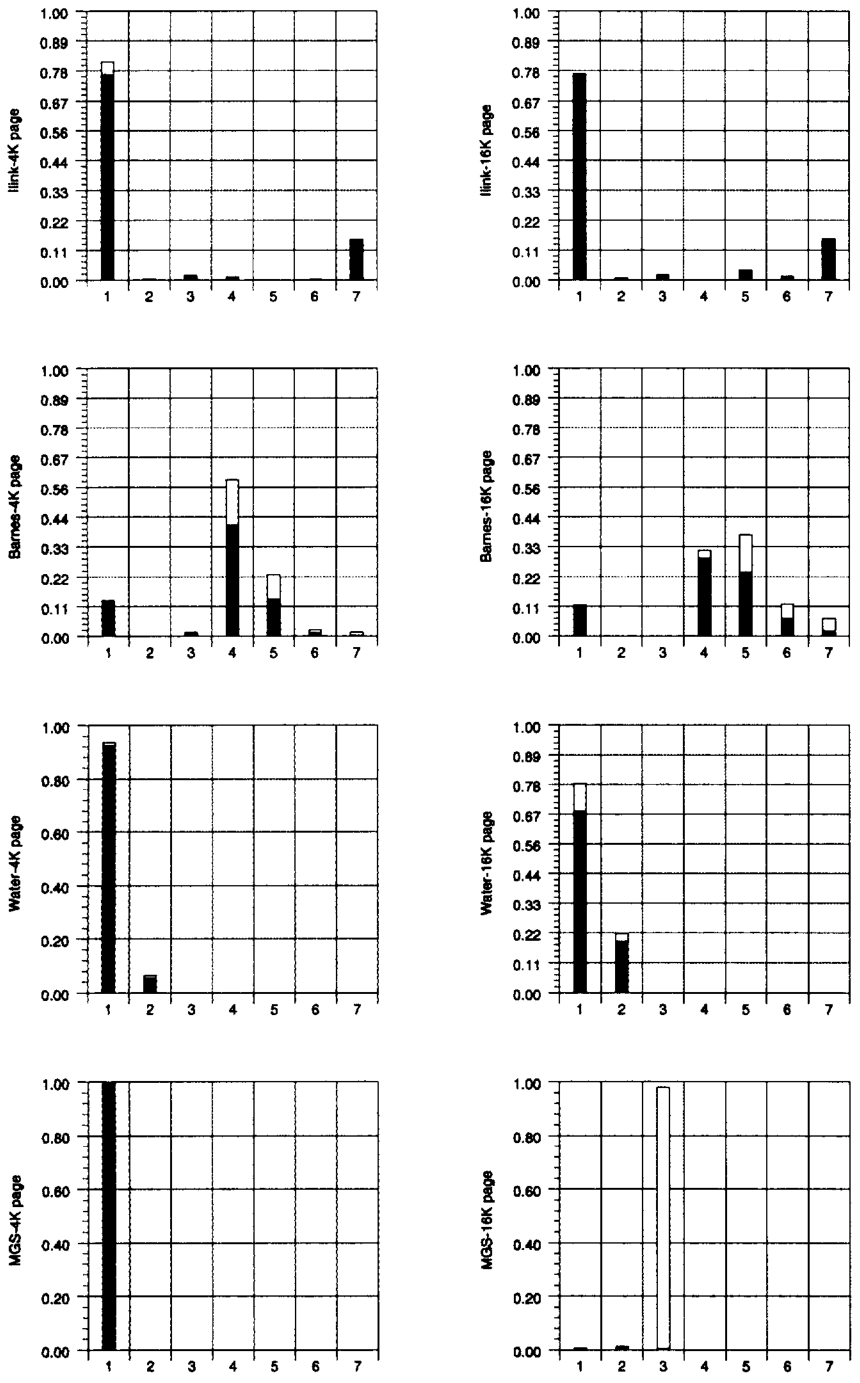

Figure 3: False sharing signature for 4 Kbyte and 16 Kbyte consistency units. On the horizontal axis, the number of requests to concurrent writers as seen at page faults. Each bar represents the frequency of requests of that type. Communication breakdown: black - useful, white - useless

With their smallest input size, all these applications have coarse-grain reads with the granularity of each read roughly matching the 4 Kbyte page size. This is demonstrated by the absence of piggybacked useless data at the 4 Kbyte page size. Thus, by increasing the consistency unit, we may introduce piggybacked useless data, or, worse, we may introduce useless messages, resulting from increase in the number of concurrent writers (a shift in the false sharing signature).

In all cases, the dynamic aggregation algorithm performs very well. For all applications and data sets, its performance is at worst a few percent below that of the best choice of the static consistency unit size.

5.5 Detailed Discussion of the Applications

Barnes 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 tree is constructed sequentially by a master processor, and the force computation is done in parallel by all processors. Reads and writes are to small individual particle data structures, with fine granularity. The fine-grained writes lead to a high write-write false sharing pattern for all pages on which bodies are allocated. However, there is also extensive true sharing, and thus there are relatively few useless messages. The master processor reads essentially the entire region of memory in which bodies are allocated. All other processors also read parts of the body array according to the partition computed by the master processor. Useful messages typically carry a large amount of useless data. Aggregation is beneficial because of the large region of memory accessed by each processor.

Ilink is a genetic linkage analysis program that locates specific disease genes. The main data structure is a pool of sparse arrays called **genarrays**. The sharing pattern is master/slave. The master processor assigns the non-zero 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 write to it concurrently. Both the read and write granularity are very small, and there is extensive write-write false sharing. All processors access every page of the **genarrays**. The master processor reads the **genarrays** from all the slaves, and, afterwards, all slaves read them from the master. This behavior explains the shape of the false sharing signature, with most messages falling either into the 1 or the 7 category, and very few useless messages. Aggregation is beneficial for Ilink, for the same reason as in Barnes. A large region is accessed, and little false sharing is added by going to a larger consistency unit.

TSP uses a branch-and-bound algorithm to find the minimum cost tour. The major shared data structures are the pool of partially evaluated tours, a priority queue containing pointers to tours in the pool, and the current shortest path. All the major data structures migrate among the processors. Both the useless messages and data result from bringing in diffs corresponding to tours allocated by other processors but not read by the faulting processor. The tour pool and the priority queue are both multi-page data structures, and accesses to these data structures are scattered and irregular. Aggregation thus reduces the number of messages with an improvement in the execution time.

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 array of molecules is shared, allocated contiguously, and partitioned among processors. A lock protects access to each molecule. Write-write false sharing occurs in the intra-molecular force computation phase, at the boundaries between regions owned by different processors. In the inter-molecular force computation phase, each processor computes and updates the force between each of its molecules and each of $n/2$ molecules following it in the array in a wrap-around fashion. Thus, although each read is fine-grained (one molecule), the region read by each processor covers half the shared array. There may also be write-write false sharing in this phase, if two processors update molecules colocated on the same page. However, this is highly unlikely because of the staggered update pattern. Useless messages occur mostly in the intra-molecular phase, when a processor receives data for molecules that belong to the preceding neighbor's region. Useless messages can also occur in the inter-molecular computation phase when the end of the region read by a processor falls on a write-write falsely shared page. In this case, requests for updates may be sent to two processors when only one request is useful. For larger page sizes, this is more likely to happen, as shown in the false sharing signature in Figure 3. Overall, there is a slight increase in the number of useless messages when going to larger consistency units. Other than useless data carried in useless messages, private data in each molecule data structure causes a large amount of useless data carried in useful messages. Aggregation is beneficial because of the large region accessed by each processor. The hysteresis effect of the dynamic scheme, while it adapts to a new access pattern, causes the extra useless messages compared to the 4 Kbyte page size.

Jacobi solves a differential equation on a square grid. Each processor is assigned a band of rows. The boundary row is communicated between neighboring processors. The pages containing the boundary row are entirely written, and therefore communicated. If the pages contain private data in addition to the boundary row, this data becomes useless data. However, there are never useless messages, because even if there is false sharing at the boundary, there is always true sharing on those pages as well. With a 4 Kbyte consistency unit, there is no useless data for the 1Kx1K data set, but there is useless data for the 8 and 16 Kbyte consistency unit, causing a very slight degradation in performance. Similarly, there is useless data in going from 8 to 16 Kbytes for the 2Kx2K data set.

3D-FFT numerically solves a partial differential equation using forward and inverse FFT's with a transpose being performed to optimize the computation. Each processor is assigned a contiguous section of the array. Communication occurs in performing the transpose, and is of a producer-consumer nature. During the transpose, a processor may receive diffs for a full page of which it only reads a part. The other parts written that are communicated are useless data piggybacked on a useful messages. A small data structure used for a checksum computation is concurrently written by all processors. This data structure causes a few useless messages because one master processor reads it while all other processors only write it. However, because there is only one page that exhibits this pattern, the fraction of useless messages is low for non-trivial problem sizes, and the useless data in these messages is of the order of a few bytes. If, for a particular data set size, the consistency units communicated during the transpose contain more than the part that each processor is reading, there is a substantial increase in the amount of useless data. For example, for the 64x64x64 problem size, each processor reads data at an 8 Kbyte granularity. This explains both the aggregation effect when going from 4 to 8 Kbyte pages, and the piggybacked useless data that occurs when the data for the full 16 Kbyte consistency unit is transferred. As a result, performance improves from 4 to 8 Kbytes, but deteriorates when going to 16 Kbytes. This effect can be seen also for the 64x64x32 data set when going from 4 to 8 and 16 Kbyte consistency units.

MGS computes an orthonormal basis for a set of N-dimensional vectors. The vectors are divided between processors in a cyclic manner. In each iteration, the program contains two phases. In the first phase, every processor computes its own pivot vector followed by a barrier. Afterwards, every processor makes all vectors that follow the pivot vector orthogonal to the pivot vector. Because of the cyclic distribution, a processor's write granularity is the size of a vector. The read granularity is the same (the pivot vector is read). With the 1Kx1K input size the read and write granularity correspond exactly to the size of the 4 Kbyte consistency unit. When we increase the size of the consistency unit to 8 or 16 Kbytes, two and four vectors respectively become colocated on the same page. Thus, each page in shared memory is written concurrently by at least two processors for these page sizes. When the processor writes to its own vectors, it requests updates from all the concurrent writers of vectors colocated on the same page. These are useless messages and constitute most of the communication as seen from the false sharing signature in Figure 3. Furthermore, when reading the pivot row, a processor receives useless messages from the concurrent writers of the page containing the pivot row. The increase in useless messages causes larger consistency units to perform drastically worse for MGS. The dynamic scheme performs the same as the static 4 Kbyte page. There is no repetition in any processor's data fetch pattern, and thus no gain from aggregating pages in page groups.

Shallow is a Fortran code that solves difference equations on a two dimensional grid, for weather prediction. Each processor is allocated a chunk of columns from a set of arrays. There is write-write false sharing at some interprocessor boundaries for the shared arrays. Whenever data written by another processor is required, only the one column at the boundary is needed. So when the boundary page contains two columns, this may result in either piggybacked useless data to be sent or useless messages. These two patterns occur independently on different arrays. For some of the arrays, processors write only to their own columns, and read the first column of the right neighbor's chunk. Thus, the pattern is similar to Jacobi and piggybacked useless data occurs for large page sizes. For some other arrays, however, processors write to some of their own columns, but also write to the first column of the right neighbor's chunk. On the other hand they do not read any of the neighbor's columns. Thus, the write-write false sharing causes useless messages if a page becomes large enough to hold two columns. Apart from the above patterns, there is a wrap-around copy pattern where a master processor copies the last column of a particular array to the first column. Only piggybacked useless data results from these accesses. Because of the rather complex pattern, going to 8 and 16 Kbyte consistency units for the smallest problem size, we see both an aggregation effect, decreasing the number of useful messages, and a false sharing effect, increasing the number of useless messages and piggybacked useless data. Overall, the combination results in a slight decrease in performance. For larger data sets, false sharing subsides, and aggregation causes performance to improve.

6 Related Work

There are both compile-time and run-time schemes to eliminate false sharing. The Midway DSM system [19] supports the entry consistency model, which requires the programmer to associate each shared datum explicitly with a synchronization variable. False sharing can be entirely avoided, because the consistency unit is precisely the shared data. Granston and Wijshoff [11] present compile-time techniques for a parallelizing Fortran compiler that transforms loops in order to reduce false sharing. Jeremiassen presents compile-time techniques to analyze the sharing behavior of explicitly parallel programs [12]. His analysis determines which data structures may be susceptible to false sharing. Heuristics are then applied to determine if it is profitable to pad the data structures.

Eggers and Katz [6, 7] showed that the performance of coherent caches for bus-based shared memory multiprocessors depends on the relationship between the cache block size, the granularity of sharing, and the locality exhibited by a program. They showed that the optimal cache block size varies for different sets of applications. Gupta and Weber [18] examined the effect of cache block size on the number and size of invalidations in a directory-based cache-coherent multiprocessor system. They too noticed that different applications gave their best performances for different cache block sizes. They traced the source for this variation to the different sharing patterns among the applications.

Dubnicki and LeBlanc [4] 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 size of the 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. In an earlier study [10], Goodman also evaluated the effect of the size of the consistency units on the behavior of a virtual address cache.

Zhou et al. [20] discuss the relationship between relaxed consistency and coherence granularity in DSM systems. They conclude that sequential consistency with small consistency units and lazy release consistency with larger consistency units perform comparably for the applications used in the study. They only consider consistency units up to the size of the virtual memory page, while we study consistency units that are a multiple of the page size.

Lu et al. [15] analyzed the performance differences between message passing programs, using Parallel Virtual Machine (PVM) [8], and DSM programs, using TreadMarks. They identified lack of aggregation, separation between synchronization and data movement, false sharing and diff accumulation for migratory data as the primary factors for the poorer performance of the TreadMarks programs.

7 Conclusions

In this paper, we study the tradeoffs between false sharing and aggregation in software DSM, for consistency units larger than the hardware page size.

We first document the effects of false sharing when the virtual memory page size is used as the consistency unit. Useless messages and useless data may result. For our applications, the number of useless messages is relatively small, but the amount of useless data may be substantial.

We then increase the consistency unit to be a multiple of the page size. Paradoxically, performance improves for the applications that already exhibited a fair amount of false sharing at the virtual memory page size. The paradox is explained by a combination of two facts. First, these applications typically access a large region of shared memory, so that aggregation reduces the number of messages. Second, there is only a small increase in useless messages or useless data induced by false sharing, because there is both true and false sharing on most pages. Conversely, if there is little or no false sharing at the virtual memory page size, then performance can deteriorate when increasing the consistency unit size, sometimes dramatically if a large number of useless messages are introduced.

We finally demonstrate that it is possible to obtain most of the benefits of aggregation without the potential problems of false sharing, by using a simple dynamic aggregation algorithm. The dynamic scheme automatically aggregates pages into larger *page groups* at runtime. No user or compiler support is needed. The results approach those obtained with the best static consistency unit size.

Acknowledgments

This work is supported in part by the NSF under Grants CCR-9410457, BIR-9408503, CCR-9457770, CCR-9502500, CCR-9521735, CDA-9502791, and MIP-9521386, also by the Texas TATP program under Grant 003604-017, and by grants from IBM Corporation and from Tech-Sym, Inc. We would also like to thank the anonymous reviewers for their comments, and Ramakrishnan Rajamony for his help with the instrumentation of the applications.

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] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[3] 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.

[4] 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.

[5] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A.L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, 1994.

[6] 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.

[7] S.J. Eggers and R.H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.

[8] G.A. Geist and V.S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, pages 293–311, June 1992.

[9] 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.

[10] J.R. Goodman. Coherency for multiprocessor virtual address caches. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, October 1987.

[11] E. Granston and H. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.

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

[13] 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.

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

[15] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, June 1997. To appear.

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

[17] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):2–12, March 1992.

[18] 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.

[19] 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.

[20] Y. Zhou, L. Iftode, K. Li, J.P. Singh, B.R. Toonen, I. Schoinas, M.D. Hill, and D.A. Wood. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, June 1997. To appear.