Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory *

Sandhya Dwarkadas†, Kourosh Gharachorloo‡, Leonidas Kontothanassis§, Daniel J. Scales‡, Michael L. Scott†, and Robert Stets†

†Dept. of Comp. Science, University of Rochester, Rochester, NY 14627

‡Western Research Lab, Compaq Computer Corp., Palo Alto, CA 94301

§Cambridge Research Lab, Compaq Computer Corp., Cambridge, MA 02139

Abstract

Symmetric multiprocessors (SMPs) connected with low-latency networks provide attractive building blocks for software distributed shared memory systems. Two distinct approaches have been used: the *fine-grain* approach that instruments application loads and stores to support a small coherence granularity, and the *coarse-grain* approach based on virtual memory hardware that provides coherence at a page granularity. Fine-grain systems offer a simple migration path for applications developed on hardware multiprocessors by supporting coherence protocols similar to those implemented in hardware. On the other hand, coarse-grain systems can potentially provide higher performance through more optimized protocols and larger transfer granularities, while avoiding instrumentation overheads. Numerous studies have examined each approach individually, but major differences in experimental platforms and applications make comparison of the approaches difficult.

This paper presents a detailed comparison of two mature systems, Shasta and Cashmere, representing the fine- and coarse-grain approaches, respectively. Both systems are tuned to run on the same commercially available, state-of-the-art cluster of AlphaServer SMPs connected via a Memory Channel network. As expected, our results show that Shasta provides robust performance for applications tuned for hardware multiprocessors, and can better tolerate fine-grain synchronization. In contrast, Cashmere is highly sensitive to fine-grain synchronization, but provides a performance edge for applications with coarse-grain behavior. Interestingly, we found that the performance gap between the systems can often be bridged by program modifications that address coherence and synchronization granularity. In addition, our study reveals some unexpected results related to the interaction of current compiler technology with application instrumentation, and the ability of SMP-aware protocols to avoid certain performance disadvantages of coarse-grain approaches.

1 Introduction

Clusters of symmetric multiprocessors (SMP) provide a powerful platform for executing parallel applications. To ease the burden of programming such clusters, software distributed shared memory (S-DSM) systems support the illusion of shared memory across the cluster via a software run-time layer between the application and the hardware. This approach can potentially provide a cost-effective alternative to larger hardware shared memory systems for executing certain classes of workloads.

Most S-DSM systems use virtual memory hardware to detect access to data that is not available locally. Hence, data is communicated and kept coherent at the coarse granularity of a page (e.g., 4-16KB). Early page-based systems [13] suffered from *false sharing* that arises from fine-grain sharing of data within a page. More recent page-based systems [2, 3, 9, 12] address this issue by employing relaxed memory consistency models that enable protocol optimizations such as delaying coherence operations to synchronization points and allowing multiple processors to concurrently write to a page. Page-based systems may still experience overheads due to frequent synchronization or sharing at a fine granularity. Furthermore, the aggressive relaxed memory models and the required use of predefined synchronization primitives limit portability for certain applications developed on hardware multiprocessors [16].

As an alternative, a few S-DSM systems [15, 18] have explored supporting data sharing and coherence at a finer granularity (e.g., 64-256 bytes). Fine-grain access is supported by instrumenting the application binary at loads and stores to check if the shared data is available locally. Such systems provide the highest degree of transparency since they can correctly run all programs (or even binaries) developed for hardware multiprocessors by virtue of supporting similar memory models [16]. In addition, this approach reduces false sharing and the transmission of unnecessary data, both of which are potential problems in page-based systems. Nevertheless, page-based systems can potentially benefit from more optimized protocols and larger transfer granularities, without incurring the software checking overheads associated with fine-grain systems.

The recent prevalence of low-cost SMP nodes has led to extensions to software DSM designs for supporting shared memory across SMP clusters. The key advantage of using SMP nodes comes from supporting data sharing within a node directly via the cache coherence hardware, and only invoking the software protocol for sharing across nodes. Several studies have demonstrated significant gains from exploiting SMP-aware protocols in both coarse-grain [6, 14, 20] and fine-grain [17] S-DSM systems.

As described above, there are important trade-offs between *coarse-grain, page-based* and *fine-grain, instrumentation-based* S-DSM systems, both in the *performance* and the *generality* of

*This work was supported in part by NSF grants CDA–9401142, CCR–9702466, and CCR–9705594; and an external research grant from Digital/Compaq.

the shared-memory programming model. Even though there are a large number of papers that study each approach individually, a direct comparison is difficult due to major differences in the experimental platforms, applications, and problem sizes used in the various studies. Furthermore, only a few studies are actually based on SMP-aware protocols.

This paper presents a detailed comparison of the fine- and coarse-grain approaches based on the same hardware platform and applications. We use two mature systems, Shasta [17] and Cashmere [20], both of which are highly efficient and tuned to run on a state-of-the-art cluster of Digital AlphaServer multiprocessors connected through the Memory Channel network [8]. We study a total of thirteen applications: eight SPLASH-2 applications [21] that have been developed for hardware multiprocessors and five applications that were developed for page-based S-DSM systems. The first part of our study compares the performance of the *unmodified* applications on the two systems. This part allows us to evaluate the portability of applications developed for hardware multiprocessors and to measure the performance gap on applications developed for page-based systems. The second part of the study analyzes the performance of the same applications after modifications that improve their performance on either system. To ensure a fair comparison, we undertook this study as a collaborative effort between the Cashmere and Shasta groups.

Our study quantifies a number of expected trends. Shasta provides robust and often better performance for applications written for hardware multiprocessors and is better able to tolerate fine-grain behavior. Cashmere is highly sensitive to the presence of fine-grain synchronization, but provides a performance edge for applications with coarse-grain behavior. However, we found that the performance gap between the systems can be bridged by program modifications that take coherence granularity into account.

Our study also presents several unexpected results. One interesting result is that Cashmere, due to its SMP-aware implementation, shows very good performance on certain applications known to have a high degree of write-write false sharing at the page level. The regular data layout in these applications leads to page-aligned data boundaries across nodes, thus confining the write-write false sharing to processes on the same SMP node and avoiding the expected software overheads. Another interesting result is that fine-grain false sharing can sometimes favor Cashmere relative to Shasta, due to Cashmere's ability to delay and aggregate coherence operations. Finally, the instrumentation overheads in Shasta were more of a determining factor than we expected in a few cases. This effect is partly due to continued improvements in the Alpha compiler that lead to more efficient code, thus increasing the relative overhead of instrumentation code in some cases.

The only relevant study that we are aware of is by Zhou et al. [22], which also examines performance tradeoffs between fine- and coarse-grain software coherence. However, several critical differences between the studies lead to differing performance results and a number of novel observations in our work. Section 5 contains a detailed comparison of the two studies.

The remainder of this paper is organized as follows. Section 2 presents an overview of the two systems that we compare in this paper. The experimental environment is described in Section 3. Section 4 presents and analyzes the results from our comparison. Finally, we present related work and conclude.

2 Overview of Cashmere and Shasta

This section presents a brief overview of Shasta and Cashmere, and also discusses some portability issues for the two systems. More detailed descriptions of the systems can be found in previous papers [12, 15, 16, 17, 20].

2.1 Shasta

Shasta is a fine-grain software DSM system that relies on inline checks to detect misses to shared data and service them in software. Shasta divides the shared address space into ranges of memory called *blocks*. All data within a block is always fetched and kept coherent as a unit. Shasta inserts code in the application executable at loads and stores to determine if the referenced block is in the correct state and to invoke protocol code if necessary. A unique aspect of the Shasta system is that the block size (i.e. coherence granularity) can be different for different application data structures. To simplify the inline code, Shasta divides the blocks into fixed-size ranges called *lines* (typically 64-256 bytes) and maintains state information for each line. Each inline check requires about seven instructions. Shasta uses a number of optimizations to eliminate checks, reduce the cost of checking loads, and to batch together checks for neighboring loads and stores [15]. Batching can reduce overhead significantly (from a level of 60-70% to 20-30% overhead for dense matrix codes) by avoiding repeated checks to the same line.

Coherence is maintained using a directory-based invalidation protocol. A *home* processor is associated with each block and maintains a *directory* for that block, which contains a list of the processors caching a copy of the block. The Shasta protocol exploits the release consistency model [7] by implementing non-blocking stores and allowing reads and writes to blocks in pending states.

When used in an SMP cluster, Shasta exploits the underlying hardware to maintain coherence within each node [17]. The SMP-aware protocol avoids race conditions by obtaining locks on individual blocks during protocol operations. However, such synchronization is not used in the inline checking code, since it would greatly increase the instrumentation overhead. Instead, the protocol selectively sends explicit messages between processors on the same node for a few protocol operations that can lead to race conditions involving the inline checks. Because Shasta supports programs with races on shared memory locations, the protocol must correctly handle various corner cases that do not arise in protocols (such as Cashmere's) that only support race-free programs [16].

Messages from other processors are serviced through a polling mechanism in both Shasta and Cashmere because of the high cost of handling messages via interrupts. Both protocols poll for messages whenever waiting for a reply and on every loop backedge. Polling is inexpensive (three instructions) on our Memory Channel cluster because the implementation arranges for a single cachable location that can be tested to determine if a message has arrived.

2.2 Cashmere

Cashmere is a page-based software DSM system that has been designed for SMP clusters connected via a remote-memory-write network such as the Memory Channel [20]. It implements a multiple-writer, release consistent protocol and requires applications to adhere to the data-race-free or properly-labeled programming model [1]. Cashmere requires shared memory accesses to be protected by high-level synchronization primitives such as locks, barriers, or flags that are supported by the run-time system. The consistency model implementation lies in between those of TreadMarks [2] and Munin [3]. Invalidations in Cashmere are sent during a release and take effect at the time of the next acquire, regardless of whether they are causally related to the acquired lock.

Cashmere uses the broadcast capabilities of the Memory Channel network to maintain a replicated directory of sharing information for each page (i.e., each node maintains a complete copy of the directory). Initially, shared pages are mapped only on their associated home nodes. A page fault generates a request for an up-to-date copy of the page from the home node. For a page fault triggered by a write access, a *twin* (or pristine copy of the page) is created and the page is added to a per-processor *dirty list* (a list of all pages modified by a processor since the last release). As an optimization, Cashmere moves the page into *exclusive* mode if there are no other sharers, and avoids adding the page to the dirty list. In addition, the current writer is automatically made the new home node if the current home node is not actively writing the page (home node migration).

At a release, each page in the dirty list is compared to its *twin*, and the differences are flushed to the home node. After flushing the differences, the releaser sends write notifications to the sharers of each dirty page, as indicated by the page's directory entry. Finally the releaser downgrades write permissions for the dirty pages and clears the list. At a subsequent acquire, a processor invalidates all pages for which notifications have been received, and which have not already been updated by another processor on the node.

The protocol exploits hardware coherence to maintain consistency within each SMP node. All processors in the node share the same physical frame for a shared data page and hence see all local modifications to the page immediately. The protocol is also designed to avoid synchronization within a node whenever possible. For instance the protocol avoids the need for TLB shootdown on incoming page updates by comparing the incoming page to the twin if one exists, thereby detecting and applying only the modifications made by remote nodes. This allows concurrent modifications to the page by other processes within the node without the need for synchronization. The correctness of this approach depends on the assumption that programs are race-free.

2.3 Portability Issues

This section briefly discusses differences between Shasta and Cashmere with respect to two important portability issues: (a) the portability of applications to each software system, and (b) the portability of the underlying software system to different hardware platforms.

Application Portability. One of the key goals in the Shasta design is to support transparent execution of applications (or binaries) developed for hardware multiprocessors [16]. Shasta achieves this transparency by supporting memory consistency models that are similar to hardware systems. On the other hand, Cashmere (like virtually all other page-based systems) opts for a departure from the standard hardware shared-memory programming model in order to achieve better performance. By requiring the use of predefined high-level synchronization primitives to eliminate shared-memory races and using aggressive relaxed memory models, Cashmere can exploit numerous protocol optimizations that are especially important for page-based systems. However, this approach may require extra programming effort to achieve a correct and efficient port of applications that depend on the more general hardware shared-memory programming model.

System Portability. The Cashmere protocol makes heavy use of Memory Channel features, including broadcasting and guarantees on global message ordering. For example, broadcasting is used to propagate directory changes to all nodes. In addition, during a release operation, the processor sending write notifications does not wait for acknowledgements before releasing the lock. Rather, it relies on global ordering of messages to guarantee that causally related invalidations are seen by other processors before any later acquire operation. It is difficult to estimate the performance impact if the protocol were changed to eliminate reliance on broadcasting and total ordering, since the protocol design assumed these network capabilities. In contrast, Shasta was designed for a network that simply offers fast user-level message passing and is therefore more portable to different network architectures.

On the other hand, Shasta is tuned for the Alpha processor and requires detailed knowledge of both the compiler and the underlying processor architecture for efficient instrumentation. It is again hard to estimate the performance impact of moving the system to a significantly different processor architecture (e.g., Intel x86) where potentially a large variety of instructions can access memory.

3 Experimental Methodology

This section describes our prototype SMP cluster and the applications used in our study.

3.1 Prototype SMP Cluster

Our SMP cluster consists of four DEC Alpha Server 4100 multiprocessors connected by a Memory Channel network. Each AlphaServer 4100 has four 400MHz 21164 processors with 512MBytes of shared local memory. Each processor has 8K on-chip instruction and data caches, a 96K on-chip second-level cache (3-way set associative), and a 4MByte board-level cache (direct-mapped with 64-byte lines). The individual processors are rated at 12.1 SpecInt95 and 17.2 SpecFP95, and the system bus has a bandwidth of 1 Gbyte/s.

The Memory Channel is a memory-mapped network that allows a process to transmit data to a remote process without any operating system overhead via a simple store to a mapped page [8]. The one-way latency from user process to user process over Memory Channel is about 3.5 microseconds, and each network link can support a bandwidth of 70 MBytes/sec (with an aggregate bandwidth of 100MBytes/sec). For Shasta, the roundtrip latency to fetch a 64-byte block from a remote node (two hops) via the Memory Channel is 18 microseconds, and the effective bandwidth for large blocks is about 35 MBytes/s. For Cashmere, the roundtrip latency for fetching an 8K page is less than 600 microseconds.

3.2 Applications

We present results for thirteen applications. The first eight are taken from the Splash-2 [21] suite and have been developed and tuned for hardware shared memory multiprocessors. The applications are Barnes-Hut, LU, Contiguous LU (CLU), Ocean, Raytrace, Volrend, Water-nsquared, and Water-Spatial. Five Splash-2 applications are not used: four (Cholesky, FFT, Radiosity, Radix) do not perform well on S-DSM and one (FMM) has not been modified to run under Cashmere (but gets good speedup on Shasta).[1]

The remaining five applications have been developed and tuned for page-based S-DSM systems, and have been shown to have good performance on such systems [12, 20]. These applications are Em3d, Ilink [4], Gaussian Elimination (Gauss), Successive Over-Relaxation (SOR), and the Traveling Salesman Problem (TSP). Descriptions of these applications can be found in a previous paper [20].

4 Results

This section provides an in-depth comparison and analysis of the performance of Shasta and Cashmere. We begin by presenting our overall results for the unmodified applications, followed by a detailed analysis. We next consider minor application and system configuration modifications that improve performance on either Shasta or Cashmere.

4.1 Base Results for Unmodified Applications

Table 1 presents the sequential execution time (without any instrumentation or runtime library overhead), data set size, and memory usage for each application. Wherever possible, we use two dataset sizes — one relatively small, the other larger. This allows us to study performance sensitivity to the size and alignment of data structures relative to the coherence block size. For Shasta, the instrumentation overhead (not shown) increases the single-processor execution time from 9% to 55%, with an average overhead of 27% across all applications and data set sizes. The relative importance of this checking overhead decreases in parallel executions due to the typical increase in communication and synchronization overheads.

Figures 1 and 2 present the speedups on the two systems using their base configurations for the thirteen applications on 8 and 16 processors (two and four SMP nodes each with four processors), for the smaller and larger data sets, respectively. The base configurations used are a uniform block size of 256 bytes for Shasta, and a block size of 8192 bytes (the underlying page size) for Cashmere. The applications were run without any modifications (except for Barnes, as explained in the next section), and are built with the native C compiler (-O2 optimization level). Application processes are pinned to processors at startup. Execution times are based on the best of three runs, and speedups are calculated with respect to the times of the sequential application without instrumentation or protocol overhead.

Overall, the results in Figures 1 and 2 show that Shasta provides more robust and better performance on average for the eight Splash-2 applications (which have been developed for hardware multiprocessors). Nonetheless, Cashmere provides comparable or better performance on some of the Splash-2 applications, and performs much better than expected on a few applications that are known to exhibit a high degree of fine-grain sharing (e.g., LU, OCEAN). In addition, Cashmere provides superior performance for the other five applications, which have been developed for page-based systems. The next section provides a detailed analysis of these results.

4.2 Detailed Analysis of Base Results

This section presents a detailed comparison and analysis of the performance of the various applications on Shasta and Cashmere. To better understand the reasons for performance differences, we discuss the applications in groups based on their spatial *data access granularity* and temporal *synchronization granularity* (similar to notions used by Zhou et al. [22]). Applications with coarse-grain data access tend to work on contiguous regions at a time, while fine-grain applications are likely to do scattered reads or writes. The temporal synchronization granularity is related to the frequency of synchronization in an application on a given platform. An application has fine-grain synchronization if the average computation time between consecutive synchronization events is not much larger than the cost of the synchronization events themselves.[2]

Throughout this section, we will be referring to Figure 3, which provides a breakdown of the execution time for Shasta and Cashmere (labeled as SH and CSM, respectively) for each of the applications at 16 processors on the smaller data set. Execution time is normalized to that of the fastest system on each application and is split into Task, Synchronization, Data Stall, Messaging, and Protocol time. *Task* time includes the application's compute time, the cost of polling, and the cost of instrumentation in Shasta or page faults in Cashmere. *Synchronization* time is the time spent waiting on locks, flags, or barriers. *Data Stall* time measures the cumulative time spent handling coherence misses. *Messaging* time covers the time spent handling messages when the processor is not already stalled. Finally, *Protocol* time represents the remaining overhead introduced by the protocol.

Additional performance data is available in an extended version of this paper [5], including detailed execution statistics for the two systems (e.g., the number of messages, amount of message traffic, and various protocol-specific measurements) and the execution time breakdown for the larger data set.

4.2.1 Coarse-Grain Access and Synchronization

The applications in this group are CLU, Em3d, Gauss, SOR, TSP, and Water-nsquared. Overall, Cashmere is expected to perform better on these applications given the coarse communication and synchronization granularities.

[1] FMM is not "race-free" and requires additional synchronization to work correctly under Cashmere.

[2] Some applications that are classified as exhibiting coarse-grain synchronization in Zhou et al.'s study [22] exhibit fine-grain behavior in our study due to the faster processors of our platform.

Program	Smaller Problem (Data Set) Size	Time (sec.)	Larger Problem (Data set) Size	Time (sec.)
Barnes-Hut	32K bodies (39M)	15.30	131K bodies (153M)	74.69
LU	1024x1024, block: 16 (8M)	14.21	2048x2048, block: 32 (33M)	74.57
CLU	1024x1024, block: 16 (8M)	6.74	2048x2048, block: 32 (33M)	44.40
Ocean	514x514 (64M)	7.33	1026x1026 (242M)	37.05
Raytrace	balls4 (102M)	44.89	—	—
Volrend	head (23M)	3.81	—	—
Water-nsq	4K mols., 2 steps (3M)	94.20	8K mols., 2 steps (5M)	362.74
Water-sp	4K mols., 2 steps (3M)	10.94	8K mols, 2 steps (5M)	21.12
Em3d	64000 nodes (52M)	47.61	192000 nodes (157M)	158.43
Gauss	1700x1700 (23M)	99.94	2048x2048 (33M)	245.06
Ilink	CLP (15M)	238.05	—	—
Sor	3070x2047 (50M)	21.13	3070x3070 (100M)	28.80
TSP	17 cities (1M)	1580.10	—	—

Table 1: Problem and data set sizes and sequential execution time of applications.

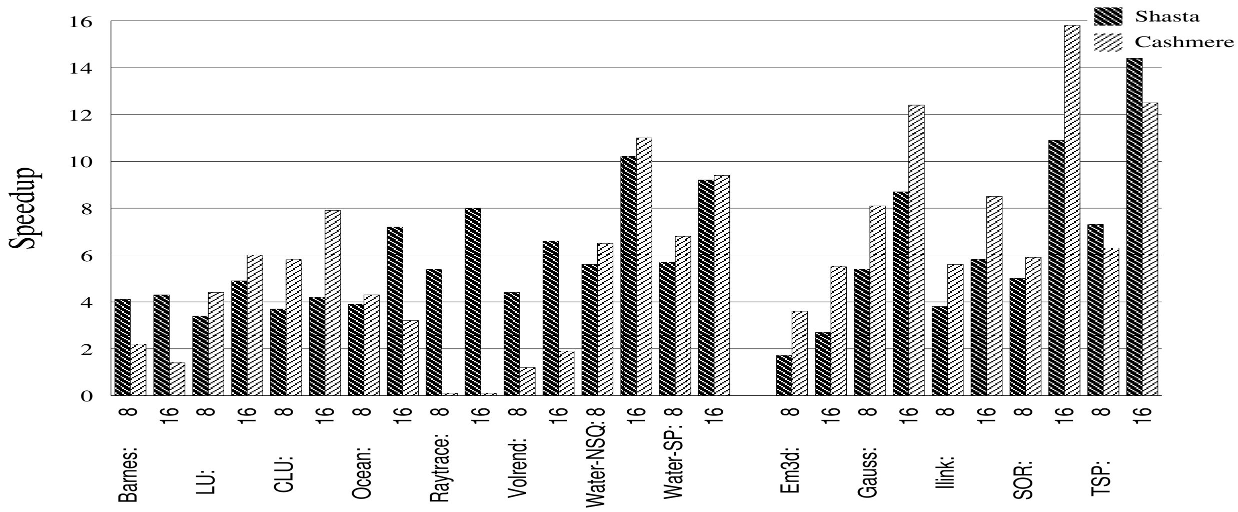

Figure 1: Speedups for the smaller data set at 8 and 16 processors.

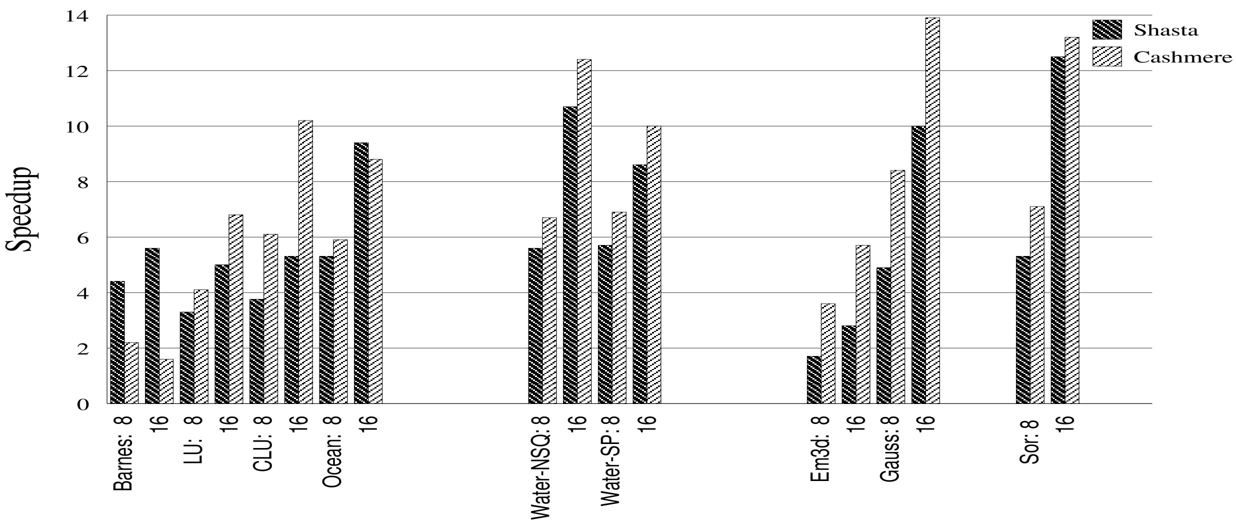

Figure 2: Speedups for the larger data set at 8 and 16 processors (Raytrace, Volrend, TSP, and Ilink not included).

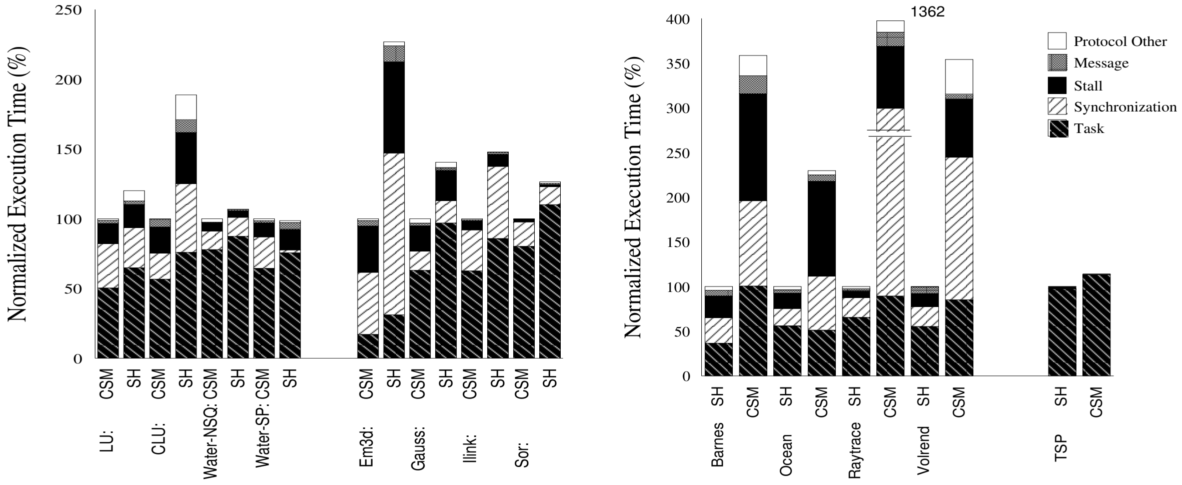

Figure 3: Application execution time breakdown on the smaller data set for 16-processor runs.

CLU uses a tiled data partitioning strategy with each tile of the matrix allocated as a contiguous chunk. Cashmere performs 1.7 times better on average than Shasta for CLU. Data is propagated more efficiently under Cashmere given its large communication granularity. As shown in Figure 3, the Cashmere data stall time component is roughly half that of Shasta. The figure also shows a higher "Task" time under Shasta because of Shasta's checking overhead (as high as 50% for a uniprocessor execution). Section 4.3 presents further results under Shasta with variable granularity and different compiler flags to address the communication granularity and checking overhead issues.

Em3d exhibits nearest-neighbor sharing, though the communication is determined at run-time based on indirection arrays. Cashmere performs two times better than Shasta on Em3d. Because of the coarse granularity of communication in this application, Shasta requires ten times more messages to fetch all the data. As we will see in Section 4.3, Shasta's variable granularity feature can be used to close this performance gap.

Gauss uses a cyclic distribution of matrix rows among processors. Cashmere performs 1.7 and 1.4 times better than Shasta for the larger dataset, and 1.5 and 1.4 times better for the smaller dataset, at 8 and 16 processors respectively. Because the matrix is triangularized, fewer elements are modified in each succeeding row, and Cashmere's large granularity causes communication of unnecessary data. For the 1700x1700 dataset, there is also write-write false sharing, since a row is not a multiple of the page size. However, the effects of false sharing are not as large as one might expect due to the SMP-aware protocol. In effect, the distribution of work becomes block-cyclic, with false sharing only on the edges of each block.

In SOR, each processor operates on a block of contiguous rows and infrequently communicates with its nearest neighbors. The determining factor is the checking overhead in Shasta (as shown by the higher "Task" times in Figure 3). Relative to Shasta, Cashmere performs 1.25 better on average.

TSP has a very coarse work granularity, so communication overheads in either system are largely unimportant. However, the more eager protocol in Shasta can lead to faster propagation of the bound value. This can in turn lead to a more efficient search (given the non-deterministic nature of the branch-and-bound algorithm). Shasta performs approximately 1.15 times better than Cashmere on TSP.

Water-nsquared partitions work such that processors modify contiguous regions of memory. Any false sharing is only at the boundaries of these regions. In addition, there is considerable node locality in the data access; at least half the lock acquires access data that was last modified within the same SMP node and is therefore fetched via the node's hardware protocol. Hence, the overheads of false sharing are small in Cashmere. Shasta's performance is primarily affected by the extra checking overhead, and Cashmere performs 1.2 times better than Shasta on average.

Overall, the coarse-grain communication and low frequency of synchronization favors Cashmere in these applications. Furthermore, for Gauss and Water-nsquared, the effect of false sharing on the performance of Cashmere is dramatically reduced by the use of SMP-aware protocols, since the false sharing largely occurs among processors on the same node. The relative performance of Shasta is often determined by the checking overhead and in some cases by its smaller data transfer granularity.

4.2.2 Fine-Grain Access with Coarse-Grain Synchronization

The applications in this group are Ilink, LU, Ocean, and Water-spatial. As we will discuss below, the use of SMP nodes with SMP-aware protocols leads to some surprising results for Cashmere.

Ilink computes on sparse arrays of probabilities and uses round-robin work allocation. The sparse data structure causes Cashmere to communicate extra data on pages that have been modified (since whole pages are communicated on a miss). Shasta's performance on Ilink is affected by three factors: the checking overhead, the small communication granularity, and the use of an eager protocol. The instrumentation overhead (as high as 60% on a uniprocessor) is due to the compiler being unable to verify the commonality of certain high-frequency double indirection operations, and Shasta therefore being unable to batch them effectively. Because the work allocation is round-robin on a per-element basis, there is also much false sharing despite the small block size used by Shasta. Shasta eagerly invalidates all copies of a block whenever any processor writes to the block and generates 10 times more protocol messages than Cashmere, which delays invalidations until synchronization points. These effects outweigh the overheads for Cashmere, and Cashmere performs 1.5 times better than Shasta.

LU uses a tiled partitioning strategy. However, unlike CLU, the matrix is allocated as a single object. Hence, each tile consists of small non-contiguous regions of memory on multiple pages. The small read granularity causes a large amount of extra data to be communicated under Cashmere. In addition, the data layout leads to a large amount of false sharing at the page level. However, LU's 2D scatter distribution leads to an assignment of tiles to processors that confines all false sharing to within each 4-processor SMP node, so all false sharing is handled in hardware. The above effect, along with the checking overheads in Shasta, allows Cashmere to perform better than Shasta by a factor of 1.2 and 1.3 times at 8 and 16 processors respectively.

Ocean also uses tiled data partitioning. For the two datasets, Cashmere performs 1.10 and 1.11 times better than Shasta at 8 processors, while Shasta is 2.25 and 1.07 times better at 16 processors. The communication is nearest-neighbor in both the column and row direction. Hence, while the tiled partitioning reduces true sharing, it increases the amount of unnecessary data communicated when a large coherence unit is used. For example, Cashmere incurs 7 to 8 times more data traffic compared to Shasta at the smaller dataset size. The extra communication generated due to false sharing in Cashmere (incurred on every boundary) increases with the number of processors, which explains Cashmere's lower relative performance at 16 processors. As in LU, the effect of false sharing in Cashmere is greatly reduced because a large portion of the false sharing is confined to individual nodes. Furthermore, both Shasta and Cashmere benefit significantly from the large portion of true sharing communication that is confined to each SMP node [17, 20].

For Water-spatial, Cashmere performs 1.2 times better than Shasta on average, with Shasta's performance being comparable to Cashmere's at the smaller dataset size and 16 processors. In this version of the fluid-flow simulation program (compared to Water-nsquared), a uniform 3-D grid of cells is imposed on the problem domain. Processors own certain cells and only access those cells and their neighbors. Molecules can move between cells during the simulation, creating a loss of locality, but this effect is small in both Cashmere and Shasta. As with Water-nsquared, the performance difference between the two systems can be attributed to the checking overhead in Shasta (as can be seen by the difference in "Task" time in Figure 3).

Overall, the performance of Shasta and Cashmere is comparable for the above set of applications. This is surprising for programs such as LU and Ocean, which exhibit frequent false sharing at the page-level. However, much or all of this false sharing turns out to occur between processors on the same SMP node (due to task allocation policy or nearest neighbor communication behavior) and is handled efficiently by the SMP-aware protocol. In addition, the low frequency of synchronization, along with the lazy protocol employed by the page-based system, allows Cashmere to tolerate any false sharing between nodes. At the same time, Shasta's eager protocol causes extra communication in applications (such as Ilink) that exhibit false sharing even at small block sizes.

4.2.3 Fine-Grain Access and Synchronization

The applications in this group are Barnes-Hut, Raytrace, and Volrend. As we will see, the combination of fine-grain data access and synchronization leads to excess communication and false sharing in page-based systems.

The main data structure in Barnes is a tree of nodes, each with a size of 96 bytes, so there is significant false sharing in Cashmere runs. Furthermore, this application relies on processor consistency in the parallel tree-building phase. Hence, while this application can run correctly on Shasta (which can enforce this form of consistency), it must be modified for Cashmere by inserting an extra flag synchronization in the parallel tree-building phase. The performance presented is for the unmodified Barnes program under Shasta, and with the additional flag synchronization under Cashmere. Shasta performs 2 and 3.5 times better than Cashmere at 8 and 16 processors, respectively. The main reason for this difference is the parallel tree building phase. This phase constitutes 2% of the sequential execution time, but slows down by a factor of 24 under Cashmere because of the fine-grain access, excessive false sharing, and extra synchronization. Shasta also suffers a slowdown in this phase, but only by a factor of 2.

Raytrace shows an even more dramatic difference between the performance of Shasta and Cashmere. Shasta performs 7 and 12.5 times better than Cashmere (which actually has a large slowdown) at 8 and 16 processors, respectively. This result is surprising, since there is little communication in the main computational loop that accesses the image plane and ray data structures. The performance difference can be primarily attributed to a single critical section used to increment a global counter in order to identify each ray uniquely. It turns out the ray identifiers are used only for debugging and could easily be eliminated (see Section 4.3.2). Their presence, however, illustrates the sensitivity of Cashmere to synchronization and data access granularity. Although only a single word is modified within the critical section, an entire page must be moved back and forth among the processors. Shasta's performance is insensitive to the synchronization, and is more in line with the behavior of a hardware DSM platform.

Volrend partitions its image plane into small tiles that constitute a unit of work, and relies on task stealing via a central queue to provide load balance. Shasta performs 3.5 times better than Cashmere for this application. Figure 3 shows that data wait and synchronization time account for 60% of the Cashmere execution time, but only about 35% of Shasta's execution. This difference results from the high degree of page-level false sharing present in the application's task queue and image data. As a result of the false sharing, Cashmere communicates over 10MB of data, as opposed to only 2MB in Shasta. The higher amount of data communication in Cashmere leads to more load imbalance among the processes, thereby triggering more task stealing that compounds the communication costs.

Overall, applications in this category exhibit by far the largest performance gap between the two systems, with Cashmere suffering considerably due to the frequent synchronization and communication.

	selected data structure(s)	block size (bytes)
LU	matrix array	2048
CLU	matrix block	2048
Volrend	opacity, normal maps	1024
EM3D	node and data array	8192
ILINK	all data	1024

Table 2: Variable block sizes used for Shasta.

4.3 Performance Improvements through Program Modifications

The performance results presented in the previous section were for unmodified programs (except to eliminate a race in Barnes for Cashmere) that were taken from either the hardware or software shared memory domain. In most cases, better performance can be achieved by tailoring the application to the latencies and granularity of the underlying software system. In this section, we present the performance of some of the applications that have been modified for either Shasta or Cashmere.

Figure 4 presents the speedups for the modified applications along with the unmodified results for 16 processor runs with the large dataset sizes. The corresponding execution time breakdowns are not shown here, but are available in an extended version of the paper [5].

4.3.1 Modifications for Shasta

The modifications we consider for Shasta are guaranteed not to alter program correctness, and can therefore be applied safely without a deep understanding of the application. This is consistent with Shasta's philosophy of transparency and simple portability. The three types of changes we use are variable granularity hints [15], the addition of padding in data structures, and the use of compiler options to reduce instrumentation overhead.

For variable granularity hints, we use a special shared-memory allocator provided by Shasta that allows one to specify the block size for the corresponding region of memory. By allocating certain regions in this manner, the application can cause data to be fetched in large units for important data structures that are accessed in a coarse-grain manner or are mostly-read. Table 2 lists the applications that benefit from using variable granularity, the data structures on which it was used, and the increased block size. As an example of the benefit of variable granularity, the performance of EM3D improves by a factor of 1.8 and CLU by a factor of 1.2 for the large input set on 16 processors (results labeled as "SH-VG" in Figure 4).

Another change that can sometimes improve performance is padding elements of important data structures. For example, in the Barnes-Hut application, information on each body is stored in a structure which is allocated out of one large array. Since the body structure is 120 bytes, there is some false sharing between different bodies. Shasta's performance improves significantly (by a factor of 1.9 on the large input set for 16 processors) by padding the body structure to 128 bytes (labeled as "Padded" in Figure 4).

A final modification involves using compiler options to reduce instrumentation overhead. Existing compilers typically unroll inner loops to improve instruction scheduling and reduce looping overheads. Batching of checking code is especially effective for unrolled loops, since unrolling increases the number of neighboring loads and stores in the loop bodies. In CLU, instrumentation overhead is still high despite the batching, because the inner loop is scheduled so effectively. The checking overhead is reduced significantly (from 55% to 36% on a uniprocessor with the large input set) by using a compiler option that increases the unrolling of the inner loop from the default four iterations to eight iterations (labelled as "Loop Unroll" in Figure 4).

There are a large class of other optimizations that would improve application performance under Shasta. However, we have limited our investigation to simple hint optimizations to emphasize the ease of portability of applications from hardware multiprocessors to Shasta.

4.3.2 Modifications for Cashmere

The modifications made to tune the applications for Cashmere aim to reduce the frequency of synchronization or to increase the granularity of sharing. Unlike the changes made for Shasta, the Cashmere modifications are not in the form of hints and require some real understanding of the application in order to maintain correctness. We have made changes to three applications that exhibit particularly poor performance under Cashmere (results are labeled as "Restructured/CSM" in Figure 4).

The major source of overhead in Barnes-Hut is in the tree building phase. The application requires processors to position their bodies into a tree data structure of cells, resulting in a large number of scattered accesses to shared memory. In addition, the algorithm requires processors to synchronize in a very fine-grain manner in order to avoid race conditions. The resulting false sharing and fine-grain synchronization cause the tree-building phase to run much slower in parallel under Cashmere than in the sequential execution. While parallel tree-building algorithms suitable for page-based S-DSM [11] exist, we have chosen to use the simple approach of computing the tree sequentially (it constitutes 2% of the total sequential execution time). Building the tree sequentially does, however, have the disadvantage of increasing the memory requirements on the main node and limiting the largest problem size that can be run.

A second source of overhead comes from a parallel reduction in the main computation loop. The reduction modifies two shared variables in a critical section based on per-processor values for these variable. Performance is reduced because of critical section dilation due to page faults. We have modified the code to compute the reduction sequentially on a single processor.

Raytrace is in reality a highly parallel application. There is very little sharing and the only necessary synchronization constructs are per-processor locks on the processor work queues. However, the original version contains some additional locking code that protects a counter used for debugging purposes, as described in Section 4.2.3. Eliminating the locking code and counter update reduces the running time from 71 seconds to 3.7 seconds on 16 processors and the amount of data transferred from 1GByte down to 17MBytes. The magnitude of this improvement illustrates the sensitivity of page-based S-DSM to fine-grain synchronization.

The performance degradation in Volrend comes from false sharing on the task queue data structure as well as the small granularity of work. We have modified the application to change the granularity of tasks as well as to eliminate false sharing in the

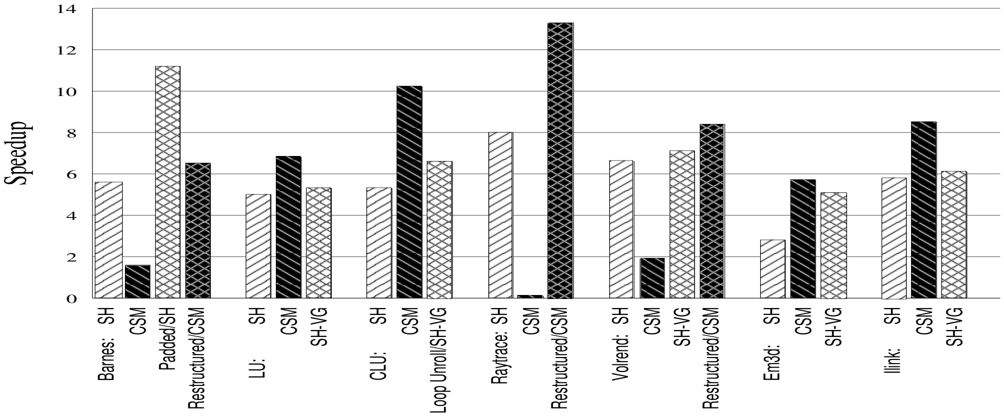

Figure 4: Speedups for the optimized applications on the large data set at 16 processors.

task queue by padding.

Additional optimizations that would improve the performance of these and other applications in our suite on a page-based system can be implemented [11]. In general, if the size of the coherence block is taken into account in structuring the application, most applications can perform well on page-based systems. Restructuring applications tuned for hardware DSM systems does, however, require knowledge of the underlying computation or data structures.

4.4 Summary of Results

This section provided an in-depth comparison and analysis of the performance of two software DSM systems, Shasta and Cashmere. We summarize our results using the geometric mean of the relative speedups on the two systems. For the eight applications (unmodified Splash-2) that were written and tuned for hardware DSM systems, Shasta exhibits a 1.6 times performance advantage over Cashmere. Most of this difference comes from one application (Raytrace) for which the performance of the two systems differs by a factor of 13. Using the same metric, for the five programs that were written or tuned with page-based DSM in mind, Cashmere exhibits a 1.3 times performance advantage over Shasta. After we allow modifications to the applications, Cashmere performs 1.15 times better than Shasta over all 13 applications. However, it is important to emphasize that the modifications we considered for Shasta were in the form of hints that do not affect application correctness or require detailed application knowledge, while the modifications we considered for Cashmere often required changes in the parallelization strategy.

5 Related Work

There is a large body of literature on S-DSM that has had an impact on the design of the Cashmere and Shasta systems. The focus of this paper is to understand the performance tradeoffs of fine-grain vs. coarse-grain software shared memory rather than to design or study a particular S-DSM system in isolation.

Iftode *et al.* [10] have characterized the performance and sources of overhead of a large number of applications under S-DSM, while Jiang *et al.* [11] have provided insights into the restructuring necessary to achieve good performance under S-DSM for a similar application suite. Our work builds on theirs by providing insight on how a similar class of programs performs under both fine-grain and coarse-grain S-DSM. In addition, we use actual systems implemented on a state-of-the-art cluster, allowing us to capture details not present in a simulation environment.

Researchers at Wisconsin and Princeton [22] have also studied the tradeoffs between fine- and coarse-grain S-DSM systems, but our studies have a number of differences. First, we have studied SMP-aware systems running on clusters of SMPs. Second, the Wisconsin/Princeton platform uses custom hardware not available in commodity systems to provide fine-grain access control. Their fine-grain performance results therefore do not include software checking overhead, which limited performance in several of our applications. In addition, the custom hardware delivers an access control fault in only 5 μs, which is fourteen times faster than the delivery of a page fault on our platform. Third, the processors in our cluster are an order of magnitude faster than those in the Wisconsin/Princeton cluster (400MHz vs 66MHz), while our network is only 3-4 times better in latency and bandwidth, thus increasing the relative cost of communication. All of these differences have manifested themselves in a number of ways in our performance results for both the fine-grain and coarse-grain systems.

Some of the results of our study mirror those of the Wisconsin/Princeton study, but others offer new insight into the granularity issue. For example, both Raytrace and Volrend perform well on the coarse-grain protocol in the Wisconsin/Princeton study, but perform very poorly on Cashmere in our study. The performance gap can be attributed to our fast hardware platform, which causes accelerated synchronization and in turn magnifies the effect of unnecessary data transferred in a coarse-grain protocol. More favorably for coarse-grain protocols, we also found that an SMP-aware implementation can greatly mitigate the effects of false sharing.

We believe that Cashmere and Shasta are among the most efficient S-DSMs in their class. There are still relatively few S-DSMs that are SMP-aware and capable of executing on commodity hardware. The Sirocco system [19] is a fine-grain S-DSM that uses an SMP-aware protocol, but its instrumentation overheads are much higher than Shasta's. SoftFlash [6] was one of the first page-based implementations designed for SMP clusters. The SoftFlash results showed that intra-node synchronization could be excessive. Cashmere-2L [20], however, combines existing techniques with a novel incoming diff operation to eliminate most intra-node synchronization. HLRC-SMP [14] is a more recent protocol that shares several similarities with Cashmere-2L. The Cashmere-2L protocol, however, has been optimized to take advantage of the Memory Channel network and allows home nodes to migrate to active writers, thereby potentially reducing twin/diff overhead.

6 Conclusions

In this paper, we have examined the performance tradeoffs between fine-grain and coarse-grain S-DSM in the context of two state-of-the-art systems: Shasta and Cashmere. In general, we found that the fine-grain, instrumentation-based approach to S-DSM offers a higher degree of robustness and superior performance in the presence of fine-grain synchronization, while the coarse-grain, VM-based approach offers higher performance when coarse-grain synchronization is used.

The performance of applications running under Shasta is most affected by the instrumentation overhead and by the smaller default block size when accessing data at a coarse granularity. Conversely, for Cashmere, the main sources of overhead are critical section dilation in the presence of fine-grain synchronization, and the communication of unneeded data in computations with fine-grain data access. Standard programming idioms such as work-queues, parallel reductions, and atomic counters can cause excessive communication overhead if they are not tuned for coarse-grain systems. However, a number of applications with false sharing at a page-level performed better than expected on Cashmere because its SMP-aware protocol enabled most or all of the false sharing effects to be handled in hardware. Finally, we found that most of the remaining performance differences between Shasta and Cashmere could be eliminated by program modifications that take the coherence granularity into account.

References

[1] S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, December 1996.

[2] 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. In *Computer*, 29(2):18-28, February 1996.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the Thirteenth SOSP*, pages 152–164, October 1991.

[4] S. Dwarkadas, R. W. Cottingham, A. L. Cox, P. Keleher, A. A. Scaffer, and W. Zwaenepoel. Parallelization of General Linkage Analysis Problems. *Human Heredity*, 44:127–141, July 1994.

[5] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. University of Rochester CS TR 699, October 1998. Also available as Western Research Lab TR 98/7.

[6] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proc. of Seventh ASPLOS*, pages 210–220, October 1996.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the Seventeenth ISCA*, pages 15–26, May 1990.

[8] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2), February 1996.

[9] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proc. of the Second HPCA*, February 1996.

[10] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory. In *Proc. of the Twenty-Third ISCA*, May 1996.

[11] D. Jiang, H. Shan, and J. P. Singh. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherence Multiprocessors. In *Proc. of the Sixth PPOPP*, June 1997.

[12] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. M. Jr., S. Dwarkadas, and M. L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proc. of the Twenty-Fourth ISCA*, June 1997.

[13] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *IEEE TOCS*, 7(4):321-359, November 1989.

[14] R. Samanta, A. Bilas, L. Iftode, and J. Singh. Home-Based SVM Protocols for SMP Clusters: Design and Performance. In *Proc. of Fourth HPCA*, pages 113–124, February 1998.

[15] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proc. of the Seventh ASPLOS*, October 1996.

[16] D. Scales and K. Gharachorloo. Toward Transparent and Efficient Software Distributed Shared Memory. In *Proc. of the Sixteenth SOSP*, October 1997.

[17] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proc. of the Fourth HPCA*, February 1998.

[18] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proc. of the Sixth ASPLOS*, pages 297–306, October 1994.

[19] I. Schoinas, B. Falsafi, M. Hill, J. Larus, and D. Wood. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Proc. of PACT '98*, October 1998.

[20] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. L. Scott. CSM-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the Sixteenth SOSP*, October 1997.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proc. of the Twenty-Second ISCA*, June 1995.

[22] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proc. of the Sixth PPOPP*, June 1997.