digital™

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

S. Dwarkadas K. Gharachorloo L. Kontothanassis D. Scales M. L. Scott R. Stets

Cambridge Research Laboratory

Cambridge Research Laboratory

Technical Report Series

CRL 98/6

April 1998

Cambridge Research Laboratory

The Cambridge Research Laboratory was founded in 1987 to advance the state of the art in both core computing and human-computer interaction, and to use the knowledge so gained to support the Company's corporate objectives. We believe this is best accomplished through interconnected pursuits in technology creation, advanced systems engineering, and business development. We are actively investigating scalable computing; mobile computing; vision-based human and scene sensing; speech interaction; computer-animated synthetic persona; intelligent information appliances; and the capture, coding, storage, indexing, retrieval, decoding, and rendering of multimedia data. We recognize and embrace a technology creation model which is characterized by three major phases:

Freedom: The life blood of the Laboratory comes from the observations and imaginations of our research staff. It is here that challenging research problems are uncovered (through discussions with customers, through interactions with others in the Corporation, through other professional interactions, through reading, and the like) or that new ideas are born. For any such problem or idea, this phase culminates in the nucleation of a project team around a well articulated central research question and the outlining of a research plan.

Focus: Once a team is formed, we aggressively pursue the creation of new technology based on the plan. This may involve direct collaboration with other technical professionals inside and outside the Corporation. This phase culminates in the demonstrable creation of new technology which may take any of a number of forms - a journal article, a technical talk, a working prototype, a patent application, or some combination of these. The research team is typically augmented with other resident professionals—engineering and business development—who work as integral members of the core team to prepare preliminary plans for how best to leverage this new knowledge, either through internal transfer of technology or through other means.

Follow-through: We actively pursue taking the best technologies to the marketplace. For those opportunities which are not immediately transferred internally and where the team has identified a significant opportunity, the business development and engineering staff will lead early-stage commercial development, often in conjunction with members of the research staff. While the value to the Corporation of taking these new ideas to the market is clear, it also has a significant positive impact on our future research work by providing the means to understand intimately the problems and opportunities in the market and to more fully exercise our ideas and concepts in real-world settings.

Throughout this process, communicating our understanding is a critical part of what we do, and participating in the larger technical community—through the publication of refereed journal articles and the presentation of our ideas at conferences–is essential. Our technical report series supports and facilitates broad and early dissemination of our work. We welcome your feedback on its effectiveness.

Robert A. Iannucci, Ph.D.
Director

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

S. Dwarkadas[1], K. Gharachorloo[2], L. Kontothanassis[3], D. Scales[2], M. L. Scott[1], R. Stets[1]

[1] Dept. of Comp. Science
University of Rochester
Rochester, NY, 14627
sandhya,scott,stets@cs.rochester.edu

[2] DEC WRL
Digital Equipment Corporation
Palo Alto, CA, 94301
kourosh,scales@pa.dec.com

[3] DEC CRL
Digital Equipment Corporation
Cambridge, MA, 02139
kthanasi@crl.dec.com

April 1998

Abstract

With the advent of commercial symmetric multiprocessors (SMPs) and low-latency, high-bandwidth networks, clusters of SMPs provide a natural base for software distributed shared memory (S-DSM). Two distinct approaches can be used: instrumentation-based approaches that provide fine-grain coherence, and VM-based approaches that restrict the underlying coherence unit to a page, but avoid instrumentation overhead. In this paper, we compare and evaluate these approaches using two mature S-DSM systems, Shasta and Cashmere, both of which run on a DEC AlphaServer SMP cluster connected by a very low-latency, high-bandwidth, remote-write, Memory Channel network.

Our results, obtained on a 16-processor, 4-node cluster, clearly illustrate the tradeoffs between fine-grain and VM-based S-DSM. A fine-grain system such as Shasta offers an easy migration path for programs developed on a hardware DSM (H-DSM). It supports H-DSM memory models, and is better able to tolerate fine-grain synchronization. In contrast, for the same unmodified programs, Cashmere's performance is highly sensitive to the presence of fine-grain synchronization, while providing a performance edge for applications with coarse-grain synchronization. With program modifications that take coherence granularity into account, the performance gap between the two systems can be bridged. Remaining performance differences are dependent on program structure: a high degree of false sharing at a granularity larger than a cache line favors Shasta, while finer-grain false sharing and large amounts of mostly private data favors Cashmere.

CRL Technical reports are available on the CRL's web page at
http://www.crl.research.digital.com.

Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square, Building 700
Cambridge, Massachusetts 02139

1 Introduction

Hardware cache-coherent multiprocessors offer good performance and provide a simple shared-memory model of computing for application programmers. Multicomputers, or collections of networked machines, can potentially provide more cost-effective performance, but require additional programmer effort to write message-passing programs. Software distributed shared memory (S-DSM) attempts to ease the burden of programming distributed machines by presenting the illusion of shared memory on top of distributed hardware using a software run-time layer between the application and the hardware.

Early S-DSM systems [14, 16] were primarily based on virtual memory. They used pages as the unit of coherence, and used page faults to trigger copy and invalidation operations. This approach can yield excellent performance for "well-behaved" programs. Unfortunately, it tends to be very slow for programs with any fine-grain sharing of data within a single page.

Recent systems [5, 4, 9, 12] employ relaxed consistency models that allow a page to be written by multiple processes concurrently, and limit the impact of false sharing to the points at which programs synchronize. Page-based systems may still experience overhead due to synchronization, sharing at a fine granularity, and metadata maintenance. Furthermore, their use of relaxed consistency models introduces a departure from the standard SMP shared-memory programming model that can limit portability for certain applications developed on hardware DSM.

Alternatively, some S-DSM systems rely on binary instrumentation of applications to catch access faults at a fine grain [20, 19]. Such systems provide the highest degree of shared memory transparency since they can efficiently run programs developed for hardware consistency models. However, the overhead of the added code can sometimes limit performance.

This paper attempts to identify the fundamental tradeoffs between these two S-DSM approaches; coarse-grain, VM-based vs. fine-grain, instrumentation-based. Our study compares two relatively mature but very different S-DSM systems running on identical hardware that are representative of the fine- and coarse-grain approaches: Shasta and Cashmere. Both systems run on clusters of Alpha SMPs connected by DEC's Memory Channel (MC) remote-write network [8].

We compare the performance of Shasta and Cashmere on thirteen applications running on a 16-processor, 4-node AlphaServer SMP cluster connected by the Memory Channel. Eight of the applications are from the SPLASH-2 application suite and have been tuned for hardware multiprocessors, while five are existing applications that have been shown in the past to perform well on software (and hardware) shared-memory multiprocessors.

We have chosen our experimental environment as representative of the way one might build an inexpensive but high-end cluster. Four-way SMP nodes provide a sweet-spot in the price/performance spectrum, while Memory Channel is a very low-latency, high-bandwidth commercial network that connects to the industry-standard PCI interface. Our environment is quite different from that used in a recent study [23] that also examines performance tradeoffs between fine- and coarse-grain software coherence. This previous study uses custom hardware to provide fine-grain access checks with no instrumentation overhead, and uses a lower performance network (almost an order of magnitude performance difference in latency and a factor of two in bandwidth). In addition, their study uses a cluster of uniprocessors, whereas we use a cluster of SMPs and protocols designed to take advantage of SMP nodes. Our end results are surprisingly similar to the ones reported by the previous study, with a few exceptions that will be discussed in detail in Section 4. Latency, bandwidth, and access check overhead differences seem to cancel out in the two studies, resulting in the similarities.

The first part of our study looks at the performance of *unmodified* applications on the two systems. This allows us to determine the portability of applications that have been tuned for good performance on H-DSM. When possible, we present results for two different size input sets in order to determine sensitivity to the size and alignment of data structures relative to the coherence block size. The second part of the study presents and analyzes the performance of the same applications once they have been optimized to improve their performance on either Shasta or Cashmere.

Our study clearly illustrates the tradeoffs between fine-grain and VM-based S-DSM on an aggressive hardware environment. A fine-grain system such as Shasta has more robust (and often better) performance for programs developed on a hardware DSM (H-DSM). It supports H-DSM memory models, and is better able to tolerate fine-grain synchronization. Cashmere has a performance edge for applications with coarse-grain data access and synchronization. With program modifications that take coherence granularity into account, the performance gap between the two systems can be bridged. Remaining performance differences are dependent on program structure: a high degree of false sharing at a granularity larger than a cache line favors Shasta since the smaller coherence block brings in less useless data; large amounts of mostly private data favors Cashmere, since there is no virtual memory overhead unless there is active sharing. Fine-grain false sharing also favors Cashmere due to its ability to delay and aggregate protocol operations.

One surprising result of our study has been the good performance of page-based S-DSM on certain applications known to have a high degree of page-level write-write false sharing. The clustering inherent with SMP nodes eliminates the software overhead from false sharing within nodes since coherence within nodes is managed by hardware. Moreover, for applications with regular (e.g. cyclic) data layout, cross-node data boundaries can end up aligned, eliminating inter-node false sharing as well.

The remainder of this paper is organized as follows. In Section 2, we describe the implementations of the two systems that we compare in this paper. In Section 3, we describe the experimental environment as well as the applications we use in this study. In Section 4, we present and analyze the results from the comparison. In Section 5, we present related work. Finally, we summarize our conclusions in Section 6.

2 Fine and Coarse-grain Approaches to Software Shared Memory

In this section, we present an overview of the most important features of Shasta and Cashmere. Detailed descriptions of the systems and the coherence protocols used can be found in other papers [19, 21].

2.1 Shasta

Shasta is a fine-grain S-DSM that relies on inline checks to detect misses to shared data and service them in software. Shasta divides up 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, if not, invoke protocol code. 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 up the blocks into fixed-size ranges called *lines* (typically 64, 128, or 256 bytes) and maintains state information for each line in a *state table*. Each inline check requires about seven instructions. Since the static and stack data areas are not shared, Shasta does not insert checks for any loads or stores that are clearly to these areas. Shasta uses a number of other optimizations to reduce the cost of checking loads and to batch together checks for neighboring loads and stores. 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. The protocol supports three types of requests: *read*, *read-exclusive*, and *exclusive* (or *upgrade*, if the requesting processor already has the line in shared state). 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 [13] by implementing non-blocking stores and allowing reads and writes to blocks in a pending state.

When used for clusters of SMPs, Shasta uses the hardware to maintain coherence within a node. The Shasta protocol avoids race conditions by obtaining locks on individual blocks during protocol operations. Since it would greatly increase the instrumentation overhead, synchronization is not used in the inline checking code. Instead, the protocol sends explicit messages between processors on the same node for protocol operations that can lead to the race conditions involving the inline checks. Because Shasta supports programs with races on shared memory locations, processes must stall at releases until *all* pending operations on the node are complete. If programs were required to be data-race-free, this extra stall time at release time could be eliminated.

Because of the high cost of handling messages via interrupts, messages from other processors are serviced through a polling mechanism in both Shasta and Cashmere. 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 distributed shared memory system that has been designed for clusters of SMPs connected via a remote-memory-write network such as the Memory Channel. It implements a multiple-writer, release consistent protocol and requires applications to adhere to the data-race-free programming model [1]. Simply stated, shared memory accesses must be protected by locks, barriers, or flags that are explicitly visible to the run-time system. These synchronization operations can be constructed from two primitives: an acquire and a release. The former is used to gain access to shared data, while the latter grants access to shared data. The consistency model implementation lies in between those of TreadMarks [4] and Munin [5]. 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 MC network to maintain a replicated directory of sharing information for each page. The directory is examined and updated during protocol actions. Initially, shared pages are mapped only on their associated home nodes. Page faults are used to trigger requests for an up-to-date copy of the page from the home node. Page faults triggered by write accesses are also used to keep track of data modified by each node. At the time of a write fault, the page is added to a per-processor *dirty list* (a list of all pages modified by a processor since the last release). If the home node is not actively writing the page, we re-assign home-ownership to the current writer and modify the directory to point to the new home node. As an optimization, we also move the page into *exclusive* mode if there are no other sharers, and avoid adding the page to the per-processor dirty list. If the page is not in exclusive mode and the faulting processor is not on the home node, a *twin*, or a pristine copy of the page, is created. The *twin* is later used to determine local modifications.

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 notices* (notifications of a page having been modified) 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 write notices 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 node. All processors in the node share the same physical frame for a shared data page. Hardware coherence then allows protocol transactions from different processors on the same node to be coalesced, resulting in reduced data communication, as well as reduced consistency overhead. One of the novelties in the protocol is that it obviates the need for TLB shootdown by using twins and diffs on both incoming and outgoing page-update operations. The use of diffs on incoming page updates allows a node to update regions of the page that were modified by a remote node without overwriting changes being made simultaneously by a local process.

2.3 Portability Issues

Cashmere makes heavy use of Memory Channel features, including broadcasting and guarantees on global message ordering in order to minimize the cost of metadata maintenance. For example, during a release operation, the processor sending write notices 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 release operation. Broadcasting is used to propagate directory changes to all nodes. Moving away from Memory Channel would require changes to the protocol in order to eliminate reliance on broadcasting and total ordering. It is difficult to estimate the performance impact of such changes 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 once again hard to estimate the performance impact of moving the system to a significantly different processor architecture (e.g. the x86) where potentially a large variety of instructions can access memory. Cashmere has no reliance on the processor architecture as long as it supports virtual memory and can deliver a trap to the user program on a VM-access violation.

3 Experimental Methodology

Our experimental environment consists of four DEC Alpha Server 4100 multiprocessors connected with a Memory Channel network. Each AlphaServer is equipped with four 21164 processors running at 400Mhz and with 512Mbytes of shared local memory. The 21164 has 8K split instruction and data first level caches, plus a 96K three-way set-associative, on-chip second level cache. The board level cache is 4 Mbytes and the cache line size is 64 bytes. Memory Channel latency and bandwidth for point to point connections are approximately $3.5\mu sec$ and $70Mbytes/sec$ respectively, while the aggregate network bandwidth is close to $100Mbytes/sec$.

Each AlphaServer runs Digital Unix 4.0D with TruCluster v. 1.5 (Memory Channel extensions). The systems execute in multi-user mode, but with the exception of normal Unix daemons no other processes were active during the tests. In order to increase cache efficiency, application processes are pinned to processors at startup. We have used the native C compiler with optimization levels set to -O2 to compile our applications and have added a post-compilation phase that adds polling code (and other code in the case of Shasta) to the executables. The extra instrumentation step is based on Digital's ATOM tool, which has been modified to allow insertion of individual instructions. Execution times for our applications were calculated based on the best of three runs and speedups were calculated against a sequential executable that was linked without the Cashmere or Shasta libraries and does not include the post-compilation step.

3.1 Application Characteristics

We present results for thirteen applications. The first eight are taken from the Splash-2 [22] suite and have been tuned for hardware shared memory multiprocessors. Five Splash-2 applications are not used: four do not perform well on S-DSM and one (FMM) has not been modified to run under Cashmere (but gets good speedup on Shasta). The remaining five applications we use are programs that have been tuned and studied in the context of software DSM systems in the past and have been shown to have good performance for such systems.

Lu: a kernel that finds a factorization for a given matrix. The matrix is divided into square blocks that are distributed among processors in a round-robin fashion.

Contiguous Lu: another kernel that is computationally identical to Lu, but allocates each block contiguously in memory.

Ocean: an application that studies large-scale ocean movements based on eddy and boundary currents. The application partitions the ocean grid into square-like subgrids (tiles) to improve the communication to computation ratio (based on true sharing and a hardware DSM)

Raytrace: a program that renders a three-dimensional scene using ray tracing. The image plane is partitioned among processors in contiguous blocks of pixels, and load balancing is achieved by using distributed task queues with task stealing.

Volrend: an application that renders a three-dimensional volume using a ray casting technique. The partitioning of work and the load balancing method are similar to those of Raytrace.

Barnes-Hut: an N-body simulation using the hierarchical Barnes-Hut method. The computation has two distinct phases. The first phase builds a shared octree data structure based on the relative positions of the bodies. The second phase computes forces between bodies and updates the relative locations of bodies based on the force calculation.

Water-nsquared: a fluid flow simulation. The shared array of molecule structures is divided into equal contiguous chunks, with each chunk assigned to a different processor. The bulk of the interprocessor communication occurs during a phase that updates intermolecular forces using per-molecule locks, resulting in a migratory sharing pattern.

Water-spatial: a fluid flow simulation that solves the same problem as Water-nsquared. It imposes a uniform 3-D grid of cells on the problem domain and uses a linear algorithm that is more efficient for a large number of molecules. Processors that own a cell need only look at neighboring cells to find molecules that might be within the cutoff radius of molecules in the owned box. The movement of molecules in and out of cells causes cell lists to be updated, resulting in additional communication.

SOR: a Red-Black Successive Over-Relaxation program for solving partial differential equations. The red and black arrays are divided into roughly equal size bands of rows, with each band assigned to a different processor. Communication occurs across the boundaries between bands.

Gauss: a solver for a system of linear equations $AX = B$ using Gaussian Elimination and back-substitution. For load balance, the rows are distributed among processors cyclically.

TSP: a branch-and-bound solution to the traveling salesman problem that uses a work-queue based parallelization strategy. The algorithm is non-deterministic in the sense that the earlier some processor stumbles upon the shortest path, the more quickly other parts of the search space can be pruned.

Ilink: a widely used genetic linkage analysis program from the FASTLINK 2.3P package that locates disease genes on chromosomes [7]. The main shared data is a pool of sparse arrays of genotype probabilities. For load balance, non-zero elements are assigned to processors in a round-robin fashion. The computation is master-slave, with one-to-all and all-to-one data communication. Scalability is limited by an inherent serial component and inherent load imbalance.

Em3d: a program to simulate electromagnetic wave propagation through 3D objects [6]. The major data structure is an array that contains the set of magnetic and electric nodes. Nodes are distributed among processors in a blocked fashion.

4 Results

In this section, we provide an in-depth comparison and analysis of the performance of Shasta and Cashmere. Table 1 presents the sequential execution times, data set sizes, and memory usage for each of the applications. The sequential times were obtained by executing the original version of each application on one processor (i.e. with no Cashmere, Shasta, or polling overhead). Wherever possible, we use two dataset sizes — one relatively small, the other larger. Figures 1 and 2 present the speedups on the two systems using their base configurations for the thirteen applications on 8 and 16 processors, for the smaller and larger data sets, respectively. The applications were run without any modifications, and the speedups are calculated with respect to the sequential times in Table 1. The base system configurations used were a uniform block size of 256 bytes for Shasta, and a block size of 8192 bytes (the underlying page size) for Cashmere.

Table 2 includes detailed execution statistics for the applications running on both protocols. The number of lock and flag acquires, the number of barrier operations, the number of messages (including data, synchronization, and protocol) and the amount of the message traffic (including application and protocol) are reported for both protocols. For Shasta, the number of read and write misses and the number of upgrade operations are also included. Read and write misses occur when referenced data is invalid, while upgrade operations occur when a block needs to be upgraded from shared to exclusive state. For Cashmere, the number of read and write page faults and the number of twins are reported.

Figures 3 and 4 provide a breakdown of the execution time for Shasta and Cashmere for each of the applications at 16 processors. 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 and the cost of instrumentation in Shasta or the cost of page faults plus polling in Cashmere. *Synchronization* time covers the time spent waiting on locks, flags, or barriers. *Data Stall* measures the cumulative time spent handling coherence misses. *Messaging* time covers the time spent handling messages when the processor is not already stalled, and finally the *Protocol* time represents the remaining overhead introduced by the protocol.

Program	Problem Size	Time (sec.)	Problem Size	Time (sec.)
LU	1024x1024 Block: 16 (8M)	14.21	2048x2048 Block: 32 (33M)	74.57
LU-contig	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	—	—
Barnes-Hut	32K bodies (39M)	15.30	131K bodies (153M)	74.69
Water-nsq	4K mols. (3M)	94.20	8K mols. (5M)	362.74
Water-sp	4K mols. (3M)	10.94	8K mols (5M)	21.12
Sor	3070x2047 (50M)	21.13	3070x3070 (100M)	28.80
Gauss	1700x1700 (23M)	99.94	2048x2048 (33M)	245.06
TSP	17 cities (1M)	1580.10	—	—
Ilink	CLP (15M)	238.05	—	—
Em3d	64000 nodes (52M)	47.61	192000 nodes (157M)	158.43

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

4.1 Detailed Analysis of Base Results

We discuss the performance of the applications in groups based on the data structures that they use. The categories we will use are: dense-matrix computations, work-queue applications, pointer-based applications, sparse-data structure applications, and irregular data access applications. Our categorization reflects access patterns that significantly affect the relative performance of the two systems. Applications in a particular category tend to exercise and showcase the same strength or weakness for each of our systems.

4.1.1 Dense-Matrix Computations

These applications—Lu, Contiguous Lu, Ocean, SOR, and Gauss—are commonly parallelized by statically partitioning the work among the available processors. The nature of the applications is such that load imbalances are rarely an issue and static partitioning works fairly well. Furthermore, the contiguous nature of the data structures used provides good locality of reference. Traditionally, one of three methods have been used to accomplish the data and work partitioning—the use of a *block*, *cyclic*, or *tiled* partitioning strategy.

SOR is an application that uses *blocked* partitioning—a set of contiguous rows are computed on by the same processor. Given the nearest neighbor communication pattern, communication occurs only among adjacent processors across nodes, resulting in a total of 12 page fetches per iteration for Cashmere and 368 data fetches for Shasta for the small data set. The overhead of extra messages reduces Shasta's performance, resulting in Cashmere's performance being 1.25 times that of Shasta's on average.

LU and Contiguous LU (CLU) use a tiled partitioning strategy. Cashmere's performance is 1.6 and 1.9 times that of Shasta's for CLU, and 1.2 and 1.3 times that of Shasta's for LU, at 8 and 16 processors respectively. The difference between LU and CLU is the memory allocation scheme. The former allocates the matrix as one object, while the latter allocates each processor's tiles as a contiguous chunk. Both applications benefit from Cashmere's large communication granularity. Data is propagated more efficiently under Cashmere, as can be seen from the data stall time, which is roughly half the data stall time under Shasta. (See figures 3 and 4.) Performance under Shasta also suffers from a high 50% checking overhead in Contiguous LU (as measured on a uniprocessor execution). On Cashmere, CLU performs much better than LU, primarily due to a two-fold reduction in the data transferred. In LU's allocation scheme, a tile is comprised of non-contiguous pieces on multiple pages. LU therefore has a mismatch between read granularity and the size of the coherence unit, and so a large amount of extra data is communicated. The data layout also should lead to a large amount of false sharing. However, LU's 2D scatter distribution assigns tiles such that all false sharing is contained within 4-processor nodes and handled by hardware.

Ocean is also an application that uses *tiled* partitioning. Cashmere's performance is 10% better than Shasta's at

	Application	LU	CLU	Ocean	Raytrace	Volrend
Shasta	Lock/Flag Acquires (K)	0 (0)	0 (0)	1.2 (0.8)	119.9	9.2
	Barriers	129 (129)	129 (129)	328 (248)	1	3
	Read Misses (K)	50.6 (199.4)	49.8 (199.2)	28.2 (37.9)	68.2	4.8
	Write Misses (K)	24.6 (98.3)	0 (0)	0 (0)	51.3	1.1
	Upgrades (K)	0 (0)	24.6 (98.3)	26.9 (37.3)	0.2	2.0
	Data Fetches (K)	75.2 (297.7)	49.8 (199.2)	27.9 (37.7)	93.2	5.1
	Messages (K)	217.6 (797.0)	178.8 (699.6)	129.4 (164.7)	428.8	24.4
	Message Traffic (Mbytes)	26.2 (101.7)	18.5 (73.4)	11.3 (14.9)	37.6	2.1
Cashmere	Lock/Flag Acquires (K)	0 (0)	0 (0)	1.2 (0.8)	120.9	9.2
	Barriers	129 (129)	129 (129)	328 (248)	1	3
	Read Faults (K)	18.5(56.0)	3.7 (12.1)	16.0 (14.3)	134.1	1.7
	Write Faults (K)	5.6 (25.0)	1.8 (6.9)	11.0 (10.8)	143.0	7.7
	Twins (K)	0 (0)	0 (0)	0 (0)	5.8	5.5
	Page Transfers (K)	6.6 (17.6)	2.0 (6.2)	10.2 (9.3)	124.2	1.2
	Messages (K)	54.2(159.1)	24.1 (56.5)	112.0(76.5)	1659.0	46.8
	Message Traffic (Mbytes)	54.6(145.1)	16.4 (51.2)	84.3 (76.9)	1027.3	10.4
	Application	Barnes	Water-NSQ	Water-SP	Sor	
Shasta	Lock/Flag Acquires (K)	68.7(274.8)	73.9 (144.2)	0.2 (0.2)	0 (0)	
	Barriers	9 (9)	12 (12)	12 (12)	48 (48)	
	Read Misses (K)	128.9 (477.0)	121.2 (285.2)	36.2 (52.9)	9.2 (13.9)	
	Write Misses (K)	51.7 (210.0)	8.1 (14.4)	0 (0)	0 (0)	
	Upgrades (K)	112.1 (419.1)	42.6 (98.5)	15.9 (26.4)	9.2 (13.9)	
	Data Fetches (K)	180.3 (686.7)	77.7 (172.8)	34.4 (52.0)	9.2 (13.9)	
	Messages (K)	985.6 (3564.4)	556.0 (1201.1)	165.0 (254.8)	38.4 (57.3)	
	Message Traffic (Mbytes)	77.7 (289.8)	37.7 (82.7)	14.1 (21.5)	3.6 (5.4)	
Cashmere	Lock/Flag Acquires (K)	413.3(1836.3)	73.9(144.1)	0.2 (0.3)	0 (0)	
	Barriers	9 (9)	12 (12)	12 (12)	48 (48)	
	Read Faults (K)	71.3 (253.3)	14.1 (26.6)	6.0 (8.6)	0.3 (0.5)	
	Write Faults (K)	112.0 (462.9)	50.1 (123.6)	3.5 (4.8)	4.8 (6.0)	
	Twins (K)	10.4 (39.1)	5.6 (2.9)	0.3 (0.3)	0 (0)	
	Data Fetches (K)	51.0 (184.4)	5.9 (11.8)	1.8 (1.8)	0.3 (0.4)	
	Messages (K)	1185.6(4910.4)	345.0(845.1)	21.6(28.7)	10.7(13.5)	
	Message Traffic (Mbytes)	425.0 (1553)	51.2 (102.3)	14.6(24.2)	2.5 (4.0)	
	Application	Gauss	TSP	Ilink	Em3d	
Shasta	Lock/Flag Acquires (K)	57.5 (69.3)	2.5	0	0 (0)	
	Barriers	6 (6)	2	522	200 (200)	
	Read Misses (K)	488.8 (526.8)	315.7	951.8	1286.1(3853.6)	
	Write Misses (K)	67.5 (96.8)	3.5	17.4	0 (0)	
	Upgrades (K)	3.0 (5.0)	25.4	166.8	1187.3(3557.1)	
	Data Fetches (K)	216.0 (307.9)	98.5	447.9	1286.1(3853.6)	
	Messages (K)	581.4 (811.8)	670.9	2384.4	5067.3(15168.1)	
	Message Traffic (Mbytes)	73.9 (104.8)	46.7	191.0	491.4 (1471.9)	
Cashmere	Lock/Flag Acquires (K)	54.1 (65.2)	2.5	0	0 (0)	
	Barriers	6 (6)	2	512	200 (200)	
	Read Faults (K)	71.6 (78.1)	11.2	85.6	45.0 (129.8)	
	Write Faults (K)	10.1 (13.1)	8.7	29.5	41.3 (116.8)	
	Twins (K)	0 (0)	0	4.1	0 (0)	
	Page Transfers (K)	18.6 (21.0)	9.7	22.6	42.4 (123.3)	
	Messages (K)	174.3 (203.8)	103.3	232.1	348.2(967.9)	
	Message Traffic (Mbytes)	153.3 (173.4)	80.3	186.4	348.5 (1014.7)	

Table 2: Detailed statistics for Shasta and Cashmere on the smaller data set at 16 processors. Statistics for the expanded data set are listed in parentheses.

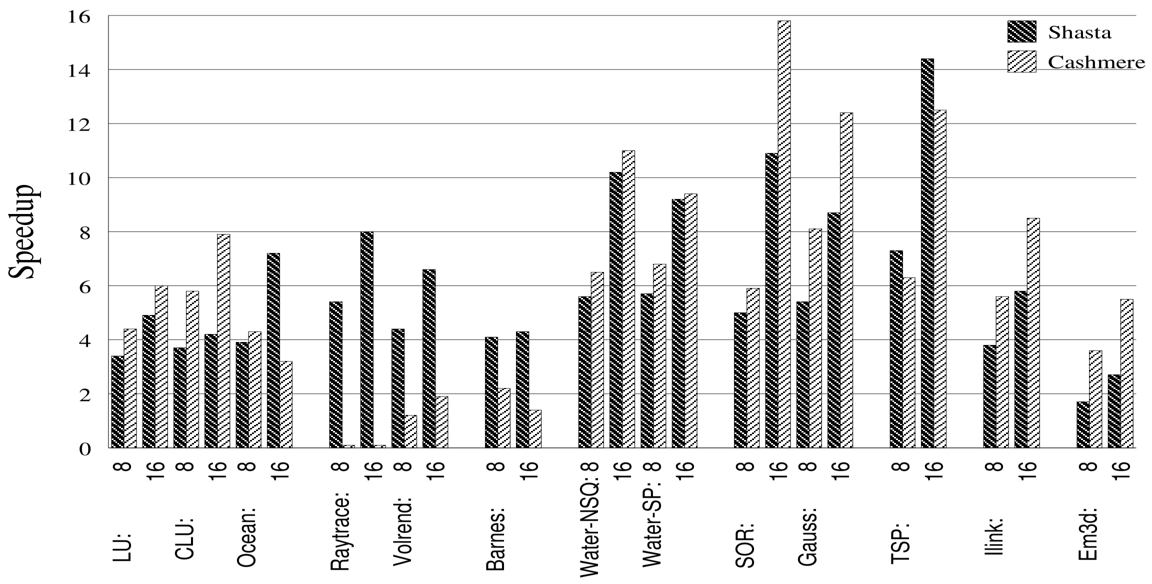

Figure 1: Speedups for the smaller data set.

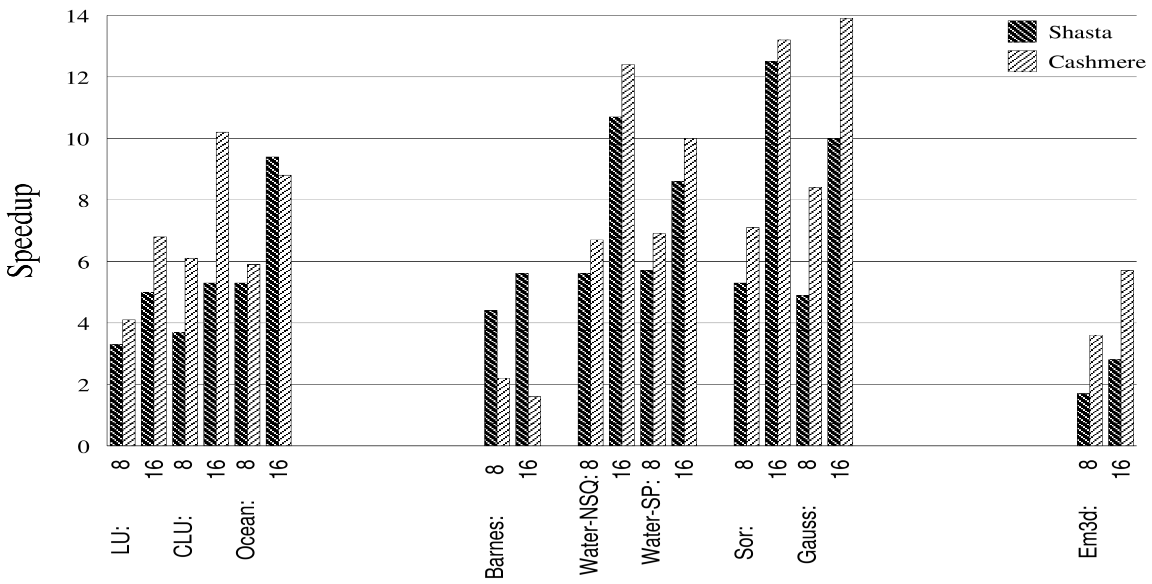

Figure 2: Speedups for the larger data set (Raytrace, Volrend, TSP, and Ilink not included).

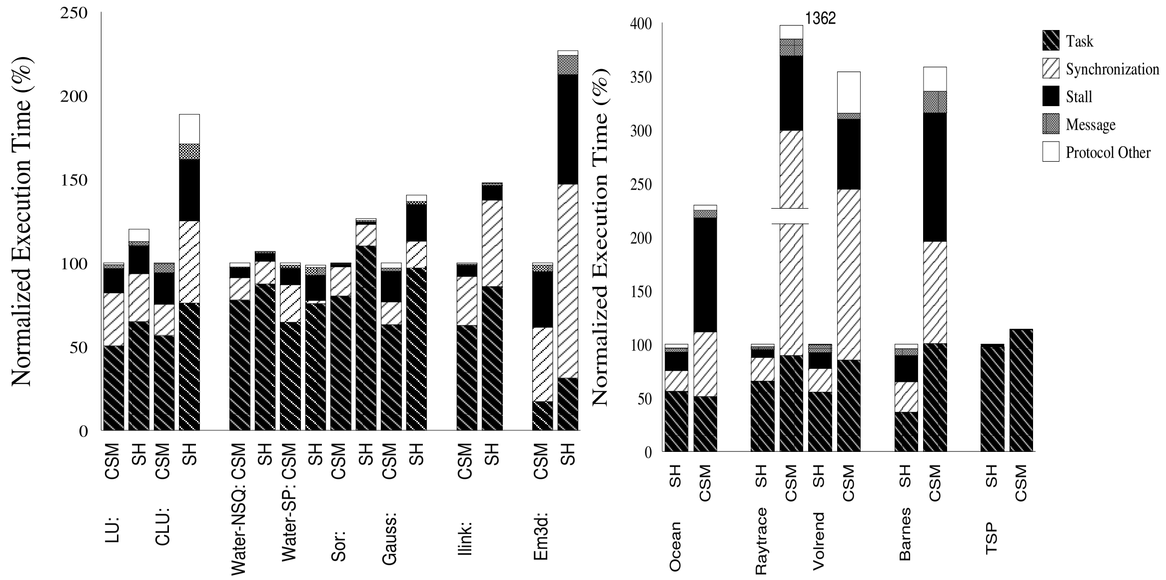

Figure 3: Application execution time breakdown on the smaller data set.

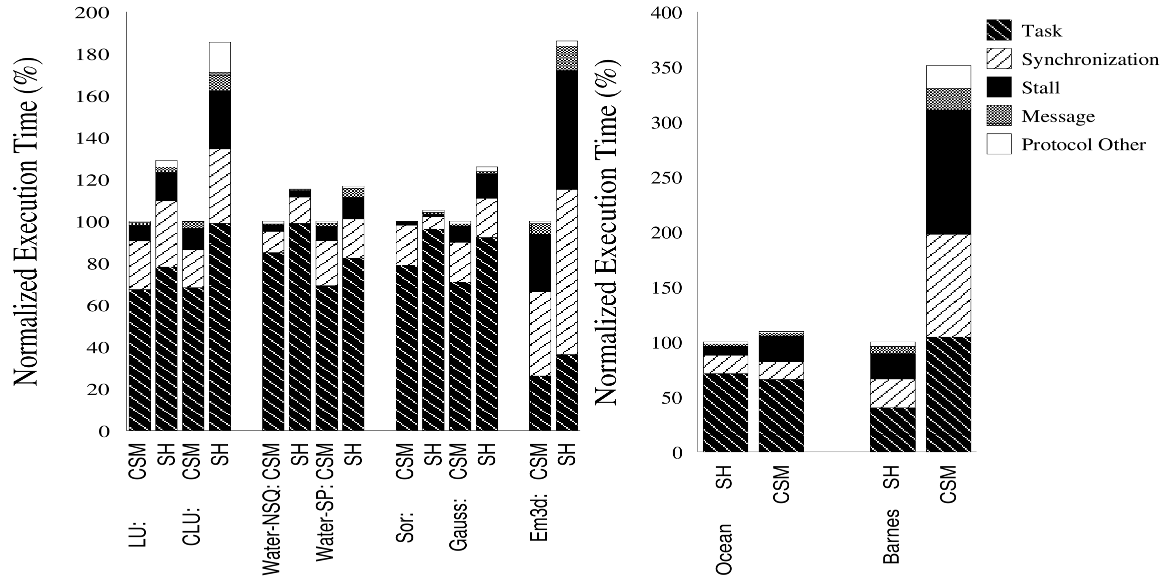

Figure 4: Application execution time breakdown on the larger data set (Raytrace, Volrend, TSP, and Ilink not included).

8 processors, and then Shasta's performance is 2.25 times better than Cashmere's at 16 processors for the 514x514 dataset. For the larger dataset (1026x1026), Cashmere performs 11% better than Shasta at 8 processors, and Shasta performs 7% better than Cashmere at 16 processors. Ocean's access patterns are 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 (84 MBytes for Cashmere versus 11 MBytes for Shasta). This overhead is offset by Shasta's instrumentation overhead, as well as the larger number of data fetches (see Figure 2). While this overhead remains constant per processor, the extra communication generated due to false sharing in Cashmere increases with the number of processors (incurred on every boundary), and hence performance is lower at 16 processors.

Gauss uses a *cyclic* distribution of matrix rows among processors. Cashmere's performance is 1.7 and 1.4 times that of Shasta's for the 2048x2048 dataset, and 1.5 and 1.4 times that of Shasta's for the 1700x1700 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 cyclic distribution of rows is used and each row of the matrix is not a multiple of the page size. Despite the use of a cyclic distribution, the effects of false sharing are not as large as one might expect due to the exploitation of SMP hardware within each node (as in LU). In effect, the distribution of work becomes block-cyclic, resulting in false sharing communication overhead only on the edges of each block. For the 2048x2048 dataset, the effect of false sharing is eliminated, since each row is a multiple of a page size, resulting in improved relative performance. Despite these effects, the overhead of the larger numbers of messages used in Shasta in transferring data (see Table 2) is higher than that caused by additional data movement in Cashmere on average.

Overall, we can see that dense matrix computation with block or cyclic distributions are very well suited for page-based DSM due to the coarse granularity of sharing and the limited amount of synchronization. Tiled distributions that do not incur false sharing (i.e. CLU) will also favor page-based DSM, while tiled distributions with smaller tiles that occupy portions of a page will incur false sharing for page-based systems and will favor the finer-grain approach.

4.1.2 Work-Queue-Based Applications

Applications in this category—Raytrace, Volrend, and TSP—are distinguished by their use of a work queue in order to partition work among processors. In these applications, such an approach is necessitated by the fact that the amount of work per data structure is not known a priori.

Raytrace is a program that shows a dramatic difference between the performance of Shasta and Cashmere. Shasta's performance is 7 and 12.5 times that of Cashmere (Cashmere shows a slowdown) at 8 and 16 processors, respectively. This result is surprising, since there is little communication in the main computational loop accessing the image plane as well as the ray data structures. The differences in performance between the two implementations can be isolated to a single critical section, which is used to access a global counter that is incremented in order to identify each ray uniquely. The ray identifiers are used only for debugging, and could easily be eliminated (we explore this option in section 4.2.2 below). 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. The performance of Shasta is insensitive to the synchronization, and is more in line with the effects one would expect on a hardware platform.

Volrend partitions its image plane into blocks, which are further subdivided into very small tiles. Each tile is a unit of work and Volrend relies on task stealing through a central queue to provide load balance. Shasta's performance is 3.5 times that of Cashmere. The performance difference stems from the application's frequent task queue synchronization. In Cashmere, the synchronization increases data movement and associated data wait time. Cashmere transfers a total of 10M of data, while Shasta transfers only about 2M (See Table 2). Also, the synchronization and data wait time account for 63% of total execution time under Cashmere, as opposed to only 36% under Shasta. The increased data movement is due to false sharing in the task queue and the image data. The frequent synchronization causes data modifications to be unnecessarily propagated. The costs of the associated page invalidations and the resulting page faults and page fetches increases data wait time and, in turn, synchronization time.

TSP is an application with a coarse work granularity. Hence, communication overheads in either system are largely unimportant. Since this application is non-deterministic, it is difficult to come to any conclusions about the differences in performance.

Overall, for applications that implement load-balancing with distributed work queues, using coarse-grain DSM can result in significant performance degradation, especially when the granularity of work is small relative to the communication time for a coherence unit. Increasing work granularity, resulting in a reduction in work-queue access (synchronization) frequency, can significantly improve the performance of page-based systems and bring them inline with that of the finer-grain approach (see Section 4.2.2).

4.1.3 Pointer-Based Applications

This category includes applications, such as Barnes, whose data structures are recursive (i.e. lists and trees) instead of flat. Such applications often incur significant amounts of false sharing in page-based systems, since node sizes are usually smaller than the size of a page. 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 had to be modified for Cashmere by inserting an extra flag synchronization in the parallel tree-building phase. The additional synchronization degrades performance even further for the page-based system.

The performance presented is for the original program under Shasta, and with the additional flag synchronization under Cashmere. Shasta's performance is 2 and 3.5 times that of Cashmere's 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 suffers a slowdown by a factor of 24 when run in parallel under Cashmere. Shasta also suffers a slowdown of a factor of 2 in this particular phase. In the case of Cashmere, this reduction in performance is a result of excess data communication due to false sharing and sparse writes in the presence of fine-grain synchronization. Shasta is far less sensitive to the presence of fine-grain synchronization due to its smaller coherence granularity, and avoids the need for the extra flag synchronization.

Overall, for applications with recursive data structures and frequent synchronization fine-grain DSM perform significantly better than their coarse-grain counterparts. The small coherence granularity results in significantly reduced sharing, data traffic, and data and synchronization stall times.

4.1.4 Sparse-Data Applications

An application in this category is ILINK, which computes on sparse arrays of probabilities and uses round-robin work allocation.

The sparsity of the data structure causes Cashmere to communicate extra data on pages that have been modified (since whole pages are communicated on a miss). In addition, the round-robin allocation of work increases false sharing.

Shasta's performance is affected by three factors—the overhead of instrumentation (which is 59% on a uniprocessor), the use of eager invalidations, and the lack of communication aggregation. The instrumentation overhead is due to the compiler being unable to verify the commonality of certain high-frequency double indirection operations, and failing to batch them as effectively as it does in other applications. The use of eager invalidations results in a large number of protocol messages (see Table 2). Since the allocation is round-robin on a per-element basis, a large number of the processors will read each coherence block, and the resulting access pattern is one-to-all (single-producer, multiple consumer). The producer subsequently has to invalidate all the copies in the consumers, resulting in the extra messages. For this application, these effects outweigh the overheads for Cashmere, resulting in Cashmere's performance being 1.5 times that of Shasta's on average.

4.1.5 Irregular Access Applications

These applications—Water-nsquared, Water-spatial, and Em3d—are distinguished by the fact that the elements that interact are determined at run-time and/or are dynamic. Hence, the amount of communication incurred cannot be statically determined.

Water-nsquared is structured so that the amount of data read remains the same. However, the amount of data written by each processor is data-dependent, though the partitioning of work results in processors modifying contiguous regions of memory. Any false sharing is only at the boundaries of these regions, and is only incurred after synchronization. There is considerable locality in the synchronization access (at least half the lock acquires access data last modified within the node). Hence, the cross-node data access in this program is coarse enough that the overheads of false sharing are small in Cashmere. Cashmere's performance is 1.2 times that of Shasta's on average. Shasta's performance is affected by the inline checks, as well as the use of small coherence blocks (increasing the number of messages used).

In Water-spatial, Cashmere also performs 1.2 times better than Shasta on average. Shasta's performance is once again affected by the inline checks, as well as the use of small coherence blocks. In this version of the fluid flow simulation program, although the processors start out working on a contiguous range of memory, the movement of molecules in and out of cells will result in a reduction in the locality of access. However, this loss in locality is small, resulting in only a small increase in the amount of data transferred for Cashmere (see Table 2) and a correspondingly negligible performance effect. The performance difference between the two systems for Water and Water-spatial can be almost exclusively attributed to the cost of instrumentation, as can be seen by the difference in the busy times in Figures 3 and 4.

Cashmere's performance for Em3d is 2 times that of Shasta's. While the access patterns are similar to those for SOR (nearest-neighbor sharing), the data structures used are more complex, and the dependencies are determined at run-time using indirection arrays. The instrumentation overhead in Shasta is therefore significant (40% for the larger dataset, and 29% for the smaller dataset). Since this application communicates data at a coarse grain among neighboring processors, Shasta sends many more messages to bring the data in, resulting in its reduction in performance (see Table 2).

Overall, the irregular applications exhibit a fair amount of false sharing but with a low frequency of synchronization. As a result, the lazy protocol employed by the page-based Cashmere system can tolerate the existence of false sharing well. The irregular nature of accesses tend to obscure access patterns and reduce the ability to coalesce access checks inserted at compile-time, thereby affecting Shasta's instrumentation overhead. In addition, the large number of data transfers in Shasta when using a small coherence block also affect performance.

4.2 Performance Improvements through Program Modifications

The performance results presented in the previous section were for programs that were taken from either the hardware or software shared memory domain and run without modifications (except to guarantee adherence to the data-race-free model in the case of Barnes). In several cases, better performance could 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 tuned for either Shasta or Cashmere. Figure 5 presents the speedups for these optimized applications, and the execution time breakdown for these applications is shown in Figure 6.

4.2.1 Modifications for Shasta

The modifications made to tune some of the applications to Shasta take the form of hints: they are guaranteed not to alter program correctness, and can therefore be applied safely without a deep understanding of the application. The three types of changes are the use of variable granularity hints, the addition of padding in data structures, and the use of compiler options to reduce instrumentation overhead.

Shasta provides a special shared-memory allocator that allows the specification of the block size for the allocated memory. This memory allocator allows data to be fetched in large units for important data structures that are accessed in a coarse-grain manner or are mostly-read. Table 3 lists the applications that benefit from using variable granularity,

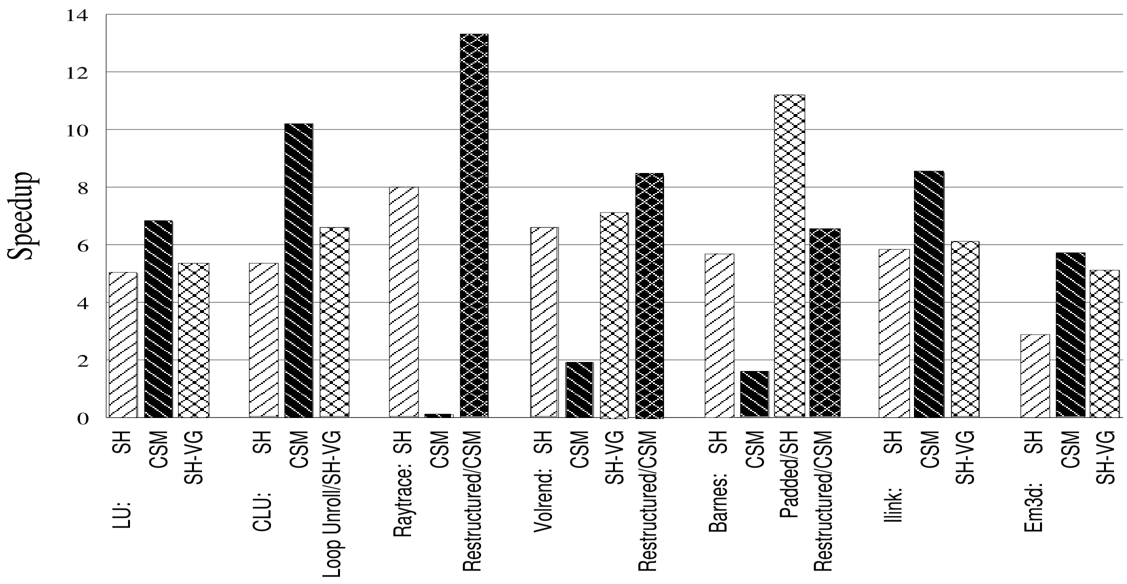

Figure 5: Speedups for the optimized applications on the large data set at 16 processors.

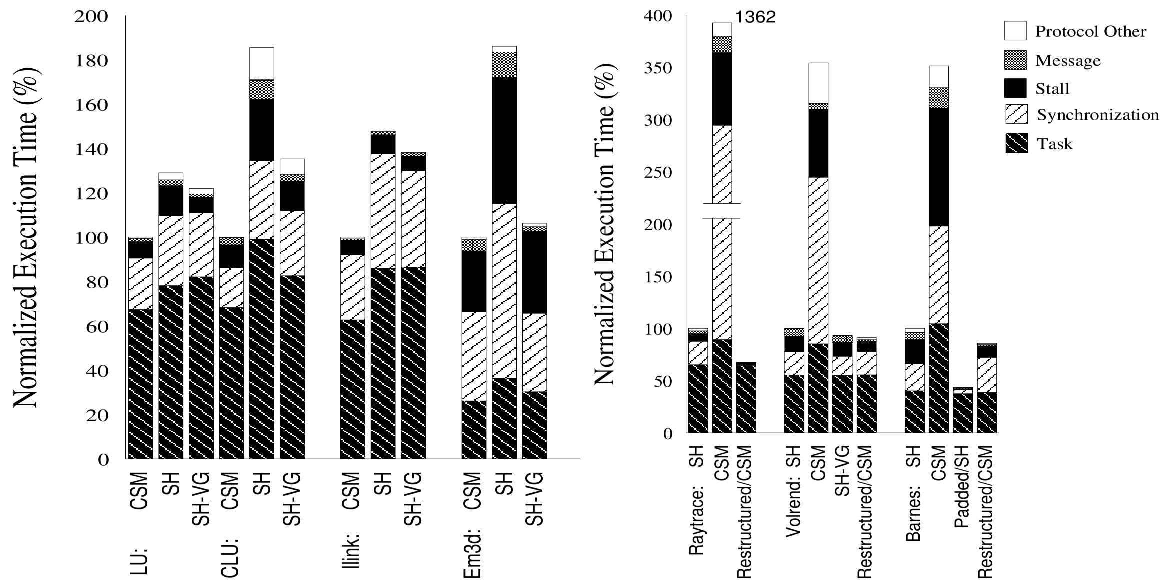

Figure 6: Execution breakdown for the optimized applications on the large data set at 16 processors.

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

Table 3: Variable block sizes used for Shasta.

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 Contiguous LU by a factor of 1.2 for the large input set on 16 processors.

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 benefits significantly (by a factor of 1.9 on the large input set for 16 processors) by padding the body structure to 128 bytes, so that false sharing is reduced.

A final modification involves using compiler options to reduce instrumentation overhead. Existing compilers typically unroll inner loops in order to improve instruction scheduling and reduce looping overheads. Batching of instrumentation is especially effective for unrolled loops, since unrolling increases the number of neighboring loads and stores in the loop bodies. In one application, LU-Contig, instrumentation overhead is still high despite the batching, because the inner loop is scheduled so effectively. The instrumentation overhead is reduced significantly (from 55% to 36% on 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.

4.2.2 Modifications for Cashmere

The modifications made to tune the applications to Cashmere aim to reduce the frequency of synchronization or to increase the granularity of sharing. They can be guided by performance profiling tools such as R_x [17] and Carnival [15]. Unlike the changes made for Shasta, they 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.

Barnes-Hut: In this application the major source of overhead is found 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. These accesses generate a large amount of false sharing. In addition, the algorithm requires processors to synchronize in a very fine-grain manner in order to avoid race conditions. The combination of a large amount of false sharing and fine-grain synchronization results in an explosion of the time it takes to build the tree. On 16 processors, building the tree consumes 36 seconds in comparison to 1.5 seconds for the sequential execution. While tree-building algorithms suitable for page-based DSM [11] exist, we have chosen to use the simple approach of computing the tree sequentially. This approach cuts our tree-building time to 2.8sec and the overall execution time to 14.3 seconds. Building the tree sequentially does have the disadvantage of increasing the memory requirements on the main node.

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. Critical section dilation due to page faults impacts performance in a major way. We have modified the code and compute the reduction sequentially on a single processor. This optimization cuts another 3 seconds out of our execution time, bringing it down to 11.5 seconds.

Raytrace: 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 additional locking code that protects a counter used only to assign a debugging field in the ray data structures. Eliminating the locking code and counter update cuts the running time from 71 seconds to 3.7 seconds on 16 processors and the amount of data transferred from 1GByte down to 17MBytes.

Volrend: The performance degradation in this application 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 task queue by padding. Our changes result in runtime improving from 2.1 seconds to 0.46 seconds. The amount of data transferred drops from 22Mbytes to 5.6Mbytes.

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 H-DSM does, however, require knowledge of the underlying computation or data structures.

4.3 Summary

We have provided an in-depth comparison and analysis of the performance of two S-DSM systems, Shasta and Cashmere. The applications were categorized on the basis of the data structures used. We summarize our results here, using the geometric mean of the relative speedup as our average performance metric.

For regular dense-matrix applications, Cashmere's average performance was 1.3 times better than Shasta's. With variable granularity modifications to the applications to aggregate communication for Shasta, the ratio drops to 1.2. The page-based approach has lower overhead for this class of applications and can better aggregate the fetch of actively shared data. For work-queue based applications, Cashmere is very sensitive to the granularity of queue access and the granularity of updates to shared counters. Shasta's average performance is 3.7 times better than Cashmere's. After the applications have been tuned, Cashmere's performance is 1.3 times that of Shasta's. For Barnes, a pointer-based application with fine-grain synchronization, Shasta's performance is 3.5 times better than Cashmere's. Comparing the best optimized versions for Shasta and Cashmere, Shasta's performance is 1.7 times better than Cashmere's. This application has a more-or-less random access pattern, and suffers under Cashmere from excess data communication due to false sharing. For Ilink, an application with sparse data structures and a fine granularity of read false sharing, Cashmere's performance is 1.5 times that of Shasta's, despite the overhead of transmitting sparsely-populated pages. Using variable granularity, Shasta closes the performance gap by 10%. Shasta loses performance because of three reasons: high instrumentation overhead, the eager invalidation of coherence blocks cached by multiple processors, and the lack of communication aggregation. Finally, for applications with irregular access patterns, Cashmere's average performance is 1.4 times better than Shasta's. If the applications are modified to take advantage of Shasta's variable granularity, Shasta exploits the same benefits as Cashmere with regard to aggregation of communication, and the ratio drops to 1.2.

Using the geometric mean for programs in our application suite that were written for H-DSM and not subsequently tuned, the fine-grain approach exhibits 1.6 times the performance of the coarse-grain approach. 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 programs that were written or tuned with page-based S-DSM in mind, the coarse-grain approach exhibits 1.3 times the performance of the fine-grain approach. With program modifications to tune the applications to the two systems, the coarse-grain approach outperforms the fine-grain approach by 15%. Modifications used by Shasta do not affect application correctness nor do they require detailed application knowledge, while modifications employed by Cashmere may require changes in parallelization strategy.

5 Related Work

There is a large body of literature on software distributed shared memory that has had an impact on the design and implementation of the Cashmere and Shasta DSM 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 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. We have also used two systems implemented on a state-of-the art cluster, which allows us to capture details not present in a simulation environment but limits our flexibility in the kind of experiments we can conduct.

Zhou [23] have also studied the tradeoffs between fine- and coarse-grain S-DSM systems. There are, however, a number of differences in our studies. We have used protocol implementations for SMP nodes, which provide different tradeoffs on how programs perform under fine- and coarse-grain coherence, and we have studied systems running on a commercially available cluster rather than systems assisted by research hardware to perform access checks. The overheads for access checks and page faults and for triggering protocol actions are much higher in our environment and could have a significant impact on the end performance of each system. In addition our network interconnect characteristics are significantly better in terms of latency and available bandwidth.

Surprisingly, with a few exceptions, our results are qualitatively similar. The differences in latency, bandwidth, and access check overheads balanced each other out and had only a limited effect on the observed performance differences between fine and coarse-grain DSM. One difference between our study and the one by Zhou *et al.* stems from the hierarchical structure of our experimental environment (i.e. cluster of SMPs) vs. the flat structure used in theirs (i.e. cluster of uniprocessors). Certain applications that exhibit a high degree of false sharing in a flat environment perform significantly better in a hierarchical setting since many coherence transactions are eliminated by being contained within a single node.

Adve *et al* [2] performed a study comparing a coarse-grain S-DSM to a region-based S-DSM. The study also analyzed the effectiveness of an instrumentation-based approach called software write detection to determine changes to data. However, the instrumentation was used only to minimize the amount of data sent during protocol actions; the coherence granularity for the systems remained at the level of a page and region, respectively.

Amza *et al* [3] present a dynamic run-time scheme to aggregate the fetch of multiple pages, and show that it is possible to obtain the benefits of aggregation without the potential problems of false sharing in the context of a page-based S-DSM. Their results show that in the presence of fine-grain false sharing, performance actually improves with the use of a larger coherence block. These results corroborate those for Ilink in Section 4.

6 Conclusions

In this paper, we have examined the performance tradeoffs between instrumentation-based and VM-based S-DSM in the context of two state-of-the-art systems: Cashmere and Shasta. In general, we have found that the instrumentation-based approach to S-DSM offers a higher degree of robustness and significantly better performance in the presence of fine-grain synchronization, while the VM-based approach offers higher performance when coarse-grain synchronization is used. Fine-grain synchronization and critical section dilation are the most significant sources of overhead for page-based S-DSM. Standard programming idioms such as work-queues, parallel reductions, and atomic counters produce extra overhead in terms of data communicated for page-based S-DSM if attention is not paid to the granularity of computation. Fine-grain, instrumentation-based S-DSM can better deal with these programming idioms because of its ability to use a smaller coherence block. However, the smaller default block-size and the instrumentation overhead can dampen performance in applications with coarse-grained synchronization and read/write granularity. In addition, applications known to suffer from false sharing on page-based systems with uniprocessor nodes can sometimes exhibit very good performance on a clustered system, since hardware maintains coherence within each SMP node.

The complementary strengths of the two approaches suggest the desirability of a hybrid system, possibly with hardware for fine-grain access checks [18, 19]. Such an approach could use fine-grain access checks to avoid unnecessary data transfers, and aggregate data communication and access checks for applications with coarse-grain access.

References

[1] S. V. Adve and M. D. Hill. A Unified Formulation of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] S. Adve, A. L. Cox, S. Dwarkadas, R. Rojamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, San Jose, CA, February 1996.

[3] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the Sixth ACM Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[4] 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*, to appear.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, October 1991.

[6] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings Supercomputing '93*, pages 262–273, Portland, OR, November 1993.

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

[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 *Proceedings of the Second International Symposium on High Performance Computer Architecture*, San Jose, CA, February 1996.

[10] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the twenty-third International Symposium on Computer Architecture*, Philadelphia, PA, 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 *Proceedings of the Sixth ACM Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[12] L. I. Kontothanassis and M. L. Scott. Issues in Software Cache Coherence. In *Fourth Workshop on Scalable Shared Memory Multiprocessors*, Chicago, IL, April 1994. Held in conjunction with ISCA '94.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[14] K. Li, P. Hudak, K. Li, J. F. Naughton, and J. S. Plank. Low-Latency, Concurrent Checkpointing for Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–878, August 1994. Originally presented at the *Fifth ACM Symposium on Principles of Distributed Computing*, August 1986.

[15] W. Meira, Jr., T. J. LeBlanc, and A. Poulos. Waiting Time Analysis and Performance Visualization in Carnival. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia, PA, May 1996.

[16] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[17] R. Rajamony and A. L. Cox. Performance Debugging Shared Memory Parallel Programs Using Run-Time Dependence Analysis. In *ACM SIGMETRICS 97*, Seatlle, WA, June 1997.

[18] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared-Memory. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 325–336, Chicago, IL, April 1994.

[19] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1996.

[20] 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 *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, San Jose, CA, October 1994.

[21] 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 *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[22] 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 *Proceedings of the Twenty-Second International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[23] 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 *Proceedings of the Sixth ACM Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

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

S. Dwarkadas K. Gharachorloo L. Kontothanassis D. Scales M. L. Scott R. Stets

CRL 98/6
April 1998