Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation

Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Singh and Kai Li

Computer Science Department
Princeton University
Princeton, NJ 08544
{yzhou,liv,jps,li}@cs.princeton.edu

Brian R. Toonen, Ioannis Schoinas, Mark D. Hill, and David A. Wood

Computer Sciences Department
University of Wisconsin, Madison
Madison, WI 53705
{toonen,schoinas,markhill,david}@cs.wisc.edu

Abstract

During the past few years, two main approaches have been taken to improve the performance of software shared memory implementations: relaxing consistency models and providing fine-grained access control. Their performance tradeoffs, however, are not well understood. This paper studies these tradeoffs on a platform that provides access control in hardware but runs coherence protocols in software. We compare the performance of three protocols across four coherence granularities, using 12 applications on a 16-node cluster of workstations. Our results show that no single combination of protocol and granularity performs best for all the applications. The combination of a sequentially consistent (SC) protocol and fine granularity works well with 7 of the 12 applications. The combination of a multiple-writer, home-based lazy release consistency (HLRC) protocol and page granularity works well with 8 out of the 12 applications. For applications that suffer performance losses in moving to coarser granularity under sequential consistency, the performance can usually be regained quite effectively using relaxed protocols, particularly HLRC. We also find that the HLRC protocol performs substantially better than a single-writer lazy release consistent (SW-LRC) protocol at coarse granularity for many irregular applications. For our applications and platform, when we use the original versions of the applications ported directly from hardware-coherent shared memory, we find that the SC protocol with 256-byte granularity performs best on average. However, when the best versions of the applications are compared, the balance shifts in favor of HLRC at page granularity.

1 Introduction

There are two important issues in providing a coherent shared address space abstraction on a network of computers, consistency models and coherence granularity. Consistency models define how applications use the shared address space, whereas the degree of the relaxation of a consistency protocol and the granularity of coherence determine the efficiency of an implementation. This paper evaluates the performance tradeoffs of the combinations of three consistency models with four sizes of coherence granularity for software shared memory implementations on a real hardware platform.

The original shared virtual memory (SVM) proposal and prototype [20] uses the traditional virtual memory access protection mechanisms to detect access misses and implements a sequential consistency model [17]. The main advantage of the approach is that it implements shared memory entirely in software on a network of commodity workstations [19] to run applications developed for hardware shared-memory multiprocessors. A disadvantage is that it restricts the coherence granularity to be a virtual memory page size. For systems with large page sizes, false sharing and fragmentation will occur in applications with multiple writer, fine-grained access patterns.

During the past few years, two main approaches have been taken to address this problem: relaxing consistency models and providing access control at a fine granularity. Relaxed consistency models introduce additional programmer restrictions in exchange for (hopefully) better performance. Examples of relaxed consistency models include release consistency [10], entry consistency [2], scope consistency [13]. Lazy release consistency(LRC) [15] is a software implementation of release consistency which delays the coherence action until the acquire time. Most software shared systems today use LRC-based protocols [14] [11] [30] [16]. These consistency models employ sophisticated protocols to reduce false sharing and fragmentation.

An alternative approach is to preserve the simplicity of sequential consistency, but find some approach to reduce the coherence granularity. Examples of providing fine-grained access control include taking advantage architectural features such as the ECC bits to trap access faults [27], using software instrumentation for shared reads and writes [27, 26], and building special access control hardware for commodity workstations [23]. The finer the granularity, the less false sharing and fragmentation occur, and hence the less need to use relaxed models. A disadvantage of fine-grain coherence is that the smaller granularity may result in excessive misses and poor remote bandwidth.

To date, the performance tradeoffs between relaxed consistency models and coherence granularities have not been well studied. This paper attempts to understand these tradeoffs by conducting experiments on a real system. Our study focuses on coherent shared memory systems with a fixed coherence granularity ; i.e. we do not permit changing the granularity within an execution. The system uses specialized hardware support for access control, and is in this sense not quite a commodity-based software-coherent system; for example, no software instrumentation is needed for fine-grain access control. However, it allows us to use a uniform mechanism for access control, and runs the coherence protocols in software. We focus on the following two questions:

- What is the best combination of granularity and consistency protocol for different classes of applications, and how much difference does it make?
- For applications that suffer performance losses in moving to coarser granularities under sequential consistency, can the performance be regained using sophisticated consistency protocols?

We conducted our experiments on a 16 dual-processor Sun SPARCstation 20s interconnected by Myrinet and augmented with fine-grained access control hardware that supports multiple sizes of coherence granularity. We studied the combinations of three consistency protocols (sequential consistency (SC) [17], single-writer lazy release consistency (SW-LRC) [16] and home-based lazy release consistency (HLRC) [30]) with four sizes of coherence granularity. We also studied two mechanisms (polling and interrupt) to handle message arrivals for each case. Our experiments used eight real benchmarks developed for hardware shared memory systems and their variations(so total 12 applications). Our applications cover most of the interesting combinations of shared data access patterns and different synchronization methods.

Our results show that no single combination of protocol and granularity performs best for all the applications. The combination of a sequentially consistent (SC) protocol and fine granularity works well with 7 of the 12 applications. The combination of a multiple-writer, home-based lazy release consistency (HLRC) protocol and page granularity works well with 8 out of the 12 applications. For applications that suffer performance losses in moving to coarser granularity under sequential consistency, the performance can usually be regained quite effectively using relaxed protocols, particularly HLRC, except when the frequency of synchronization is high. We also find that the HLRC protocol performs substantially better than a single-writer lazy release consistent (SW-LRC) protocol at coarse granularity for many irregular applications. For our applications and platform, when we use the original versions of the applications ported directly from hardware-coherent shared memory, we find that the SC protocol with 256-byte granularity performs best on average. However, when the best versions of the applications are compared, the balance shifts in favor of HLRC at page granularity.

2 Consistency Protocols

We studied three consistency protocols including sequential consistency (SC), single-writer lazy release consistency (SW-LRC), and multiple-writer, home-based lazy release consistency (HRLC). The main rationale of using these three models is twofold. First, the sequential consistency model is supported in many hardware shared memory multiprocessors. Second, the two relaxed consistency models are the latest proposed models that perform well for page-size coherence granularity.

All protocols examined in the paper support coherence granularity at block level (64, 256, 1,024, 4,096 bytes) and they all use virtual memory mapping mechanisms to allow data from a shared virtual address space to be transparently cached at a local physical address. Specifically, when a node touches a page for the first time a page fault is generated, which causes the page to be `mmap`-ed to local memory. Depending on how coherence in managed, additional actions may be necessary to initialize the page.

Each block has a *home*. Initially, blocks are assigned home nodes statically. After the beginning of an application's parallel phase, page homes migrate to the first node that "touches" them (sometimes called *first touch*). A "touch" is a load or a store for SC and store for HLRC. Regular application LU has "touch arrays" that explicitly touch data structures to manage data layout. When subsequent nodes touch an already-touched page, they go to the original home, find out about the new home, and thereafter remember the new home. Specifically, a page's home node ID is found in the distributed table and cached in a local table.

2.1 SC Protocol

The sequential consistency model allows each coherence unit to have either a single writer or one or more readers; readers and writers never co-exist at the same time. This model is generally considered the simplest for programmers, because, informally, a read always returns the result of the most recent write.

Our sequential consistency implementation is based on the Stache protocol [24] and is similar to many directory-based hardware implementations [18]. On a miss, a request message is sent to the designated *home* node. If invalidations are required, the home node collects the acknowledgments before forwarding the data to the requesting node. When an invalidation arrives at a node, the message is processed immediately (modulo the polling/interrupt issue); read-only copies are invalidated and read-write copies are written-back to the home node and invalidated.

2.2 SW-LRC Protocol

The single-writer lazy release consistency (SW-LRC) model allows a single-writer to co-exist with multiple readers and delays the propagations of updates to shared memory to the executions of acquire operations [16]. This model is more relaxed than the sequential consistency, but less relaxed than the multiple-writer lazy release consistency protocols.

Our SW-LRC protocol uses the same timestamp-based coherence control as proposed for Lazy Release Consistency [15] but it allows only a single writable copy to co-exist with multiple read-only copies. In this protocol, a write fault causes ownership to migrate but, unlike a sequential consistency protocol, read-only copies are not invalidated. Instead, SW-LRC blocks are invalidated at the acquire execution following the coherence information (**write notices**) sent with the lock. Shared blocks are version-ed each time the ownership changes. By including the versions in the write-notices and storing them away, the SW-LRC protocol can service a read fault in an one-hop roundtrip and can also avoid unnecessary invalidations.

2.3 HLRC Protocol

The Home-based Lazy Release Consistency (HLRC) protocol implements the well-known lazy release consistency model but has several performance and implementation advantages [30]. Both HLRC and traditional LRC protocols use the same multiple-writer solution based on using "twin" and "diff" but with different update schemes. Each writer is allowed to write into its copy once a clean version of the block (twin) has been created. Changes are detected by comparing the current (dirty) copy with the clean copy (twin) and recorded in a structure called a *diff*. Updates from one copy are transferred into another copy by diff-ing the first copy, sending the diff and applying it on the second copy.

The traditional LRC implementation uses a distributed diff scheme where diffs are merged on demand in a distributed fashion [14]. To bring a copy up-to-date, diffs must be applied in the proper causal order determined using vector timestamps [15].

The HLRC multiple-writer scheme differs from LRC by having the diffs sent and applied eagerly a designated home of the block. With such a scheme the home's copy of the block is kept up-to-date and its whole content will be fetched on demand to update the other copies. Our implementation extends earlier work [30] by supporting various coherence granularities.

3 Testbed

This section describes the platform used for these experiments. The testbed consists of 16 dual-processor Sun SPARCStation 20s[1]. Each contains two 66 MHz Ross HyperSPARC processors [25]; however, our study only uses one processor on each node. Each processor has a 256 KB L2 cache and each node contains 64 MB of main memory. The cache-coherent 50 MHz MBus connects the processors and memory. I/O devices reside on the 25 MHz SBus, which connects to the MBus via a bridge. All nodes run Solaris 2.4.

Each node contains a Myrinet network interface [3], which consists of a 7-MIPS custom processor (LANai) and 128KB of memory. The LANai performs limited protocol processing and schedules DMA transfers between the network and LANai memory or LANai memory and SPARC memory. The 16 nodes used in this paper are connected with three Myrinet 8-port crossbar switches. Two ports of each switch are used to connect to other switches.

Each node also contains a Typhoon-0 card that *logically* performs fine-grain access control checks on all loads and stores by *physically* snooping memory bus transactions and exploiting inclusion [22]. When the Typhoon-0 hardware detects an access control violation, it generates an exception to the shared-memory run-time system via a special fast-exception method supported by the device driver (approximately 5 μs). All protocol processing occurs on the faulting processor. The Typhoon-0 card does not directly support messaging, but does accelerate polling for messages on the Myrinet by providing a cachable location that indicates whether a message has arrived.

With Myrinet hardware, the host (SPARC) processor and Myrinet LANai processor cooperate to send and receive data. Our communication library is based on the LANai Control Program (LCP) used in Berkeley's LAM library [6]. The host processor uses loads and stores to move small messages and headers for large messages to and from LANai memory. The LANai processor uses DMA to directly move larger messages through intermediate kernel/user buffers. This organization lowers the latency of small messages and increases the throughput and the latency for large messages.

Message latency depends upon whether *polling* or *interrupts* are used to detect message reception. With polling, the LANai communicates with the Typhoon-0 board via a dedicated signal, which sets a cacheable memory location. Applications are instrumented via executable editing to check this message reception flag on all control flow backedges. With interrupts, the LANai's hardware interrupt is translated by Solaris into a UNIX signal, which takes about 70 μs. Interrupts are disabled when the application goes into the Blizzard system, meaning that only messages that arrive asynchronously while user code is executing will pay the interrupt penalty. In general, polling results in lower latency because signals are so expensive. However, polling introduces needlessly overhead when no message is present.

A microbenchmark shows 4-, 64-, 256-, 1K- and 4K-byte messages see round-trip times of 40, 61, 100, 256 and 876 usecs. Large messages achieve bandwidths of about 17 MB/sec, with is close to the values obtained by others [7, 21].

4 Applications

To evaluate the performance of the three protocols with different sizes of coherence units, we used 8 benchmarks from SPLASH-2, including LU decomposition, Ocean, FFT, Water-Nsquared, Volrend, Water-Spatial, Raytrace, and Barnes. We have two versions for Ocean, two versions for Volrend and three versions for Barnes. Because different versions have different characterstic and thus perform differently, we consider them as different applications. So, we have total number of 12 applications.

Table 1 shows the problem sizes for 8 benchmarks and their sequential execution times.

Benchmarks	Problem Size	Sequential Execution Time (secs)
LU	1024×1024	73.41
FFT	1MB	27.257
Ocean	514×514	37.43
Water-Nsquared	4096 molecules, 3 steps	575.283
Volrend	128^3 head-scaleddown2	4.493
Water-Spatial	4096 molecules, 5 steps	898.454
Raytrace	balls4	343.76
Barnes	16384 particles	33.787

Table 1: Benchmarks, problem sizes, and sequential execution times.

LU performs the blocked LU factorization of a dense matrix. In this paper, we used a version which allocates each block contiguously in virtual memory and assigns contiguous blocks to each processor.

Ocean-Original and **Ocean-Rowwise** simulates eddy currents in an ocean basin. The former is the "contiguous" version in SPLASH-2 in which the data in each subgrid are allocated contiguously in virtual memory using a 4-d array. The latter is the modified version of the noncontiguous

[1] *Note to reviewers: We actually have 40 machines and hope to have 32-node runs for the final version.*

implementation from SPLASH-2 to partition the grid rowwise.

FFT is a high-performance FFT kernel. Matrices are distributed so that every processor is assigned a contiguous set of n/p rows, and the source and destination matrices are reversed for every transpose. In this version of FFT, each processor in a transpose reads an $\frac{\sqrt{n}}{p}$ by $\frac{\sqrt{n}}{p}$ submatrix from every other processor and writes it to its local partition of set of rows.

Water-Nsquared simulates a system of water molecules in liquid state, using an $O(n^2)$ brute force method with a cutoff radius. The water molecules are allocated contiguously in an array of n molecules, and partitioned among processors into contiguous pieces of n/p molecules each. The challenging phase for SVM happens when each processor updates its own n/p molecules and the following $(n/2-n/p)$ molecules of other processors in the array, using per-partition locks for mutual exclusion.

Volrend-Original and **Volrend-Rowwise** render three-dimensional volume data into an image using a ray casting method. The two differ only in partition of tasks. In Volrend-Original, each task is a 4 by 4 block, while Volrend-Rowwise partitions task by rows.

Water-Spatial solves the same problem as Water-Nsquared, but with different data structures and different algorithms. The 3-d physical space is broken up into cells, and every processor is assigned a contiguous cubical partition of cells together with the linked lists of molecules in them.

Raytrace renders complex scenes in computer graphics using an optimized ray tracing method. The accesses to the scene data, into which rays are shot in this program, are read only. The interesting communication occurs in task stealing using distributed task queues.

Barnes-Original, **Barnes-Partree** and **Barnes-Spatial** are irregular applications which simulate the interactions among a system of particles over a number of time steps, using the Barnes-Hut hierarchical N-body method. Barnes-Original is the "rebuild" version in SPLASH-2 which builds the tree from scratch after each computation phase. Barnes-Partree uses a new tree-building algorithm that each processor first constructs a partial tree and then everything is merged into a single global tree. Barnes-Spatial partitions the global tree spatially and assigns the spaces, instead of particles, to each processor.

5 Performance

We ran the twelve applications with the combinations of three protocols (SC, SW-LRC, HLRC), four block sizes (64, 128, 1,024 and 4,096 bytes) and two mechanisms to handle message arrivals (polling and interrupt). We first present the overall application performance results and then analyze the performance difference in detail.

5.1 Overall Performance

Figure 1 shows the speedup of each application for all combinations of the three protocols and four block sizes. Since the relative performance across protocols and granularities using an interrupt mechanism is similar to that using a polling mechanism, we only present the speedups with polling, which generally performs better. We will discuss the impact of using interrupts in Section 5.4.

In our experiments, 7 applications(LU, Ocean-Rowwise, Water-Nsquared, Volrend-Rowwise, Volrend-Original, Water-Spatial and Raytrace) achieve good performance with at least some protocol and granularity. Ocean-Original, FFT and Barnes-Partree perform poorly for all protocols and granularities, Barnes-Spatial has at best reasonable performance, and for Barnes-Original the SC protocol performs reasonably at fine grain but the LRC protocols perform poorly.

At 64-byte granularity SC generally performs better than the LRC protocols except for Volrend-Rowwise and Volrend-Original. SC outperforms the LRC protocols by 5% to a factor of two for 10 applications. This indicates that the extra overhead of the relaxed protocols is not justified by the lower levels of false sharing at fine granularities. SW-LRC performs 50% to 2 times better than the HLRC protocol at 64-byte granularity for 3 applications(Ocean-Original, Barnes-Partree and Barnes-Original) because SW-LRC has less protocol overhead.

For applications that suffer performance losses in moving to coarser granularity under sequential consistency, the results in Figure 1 show that performance can usually be regained quite effectively using relaxed protocols, particularly HLRC. The original Barnes application is the only major counter-example, due to its high frequency of synchronization and hence expensive protocol activity.

Finally, let us compare the protocols at the page (4096-byte) granularity, since that is the granularity of shared virtual memory systems. For 7 applications (Ocean-Original, Volrend-Rowwise, Volrend-Original, Water-Spatial, Raytrace, Barnes-Spatial and Barnes-Partree), both SW-LRC and HLRC protocols outperform the SC protocol dramatically with 4,096-byte block size, as we might expect. The HLRC protocol improves the SC protocol performance by more than a factor of two to four, whereas the SW-LRC protocol improves by about 40-150%. For the same 7 applications with 4,096-byte block size, the multiple-writer HLRC protocol performs 30% to 4 times better than the SW-LRC protocol, showing that multiple writer protocols are indeed very valuable for irregular applications under SVM. (All these performance differences would be larger on real SVM systems, where the overheads of access violations, i.e. page faults, are higher.)

5.2 Detailed Analysis

To understand the reasons for the performance differences, it is useful to classify the applications according to their data access patterns and synchronization behavior [29, 1, 12]. In this section, we will first describe application classifications according to the number of writers per coherence unit, spatial data access granularity and temporal synchronization granularity. We will then provide a detailed analysis for each category of applications.

5.2.1 Classification

We classify the applications according to several criteria:

- **Single writer vs. Multiple writer** Based on the number of concurrent writers on the same coherence unit we can divide the applications into single writer and multiple writer applications. Write-write false sharing occurs only for multiple writer applications.

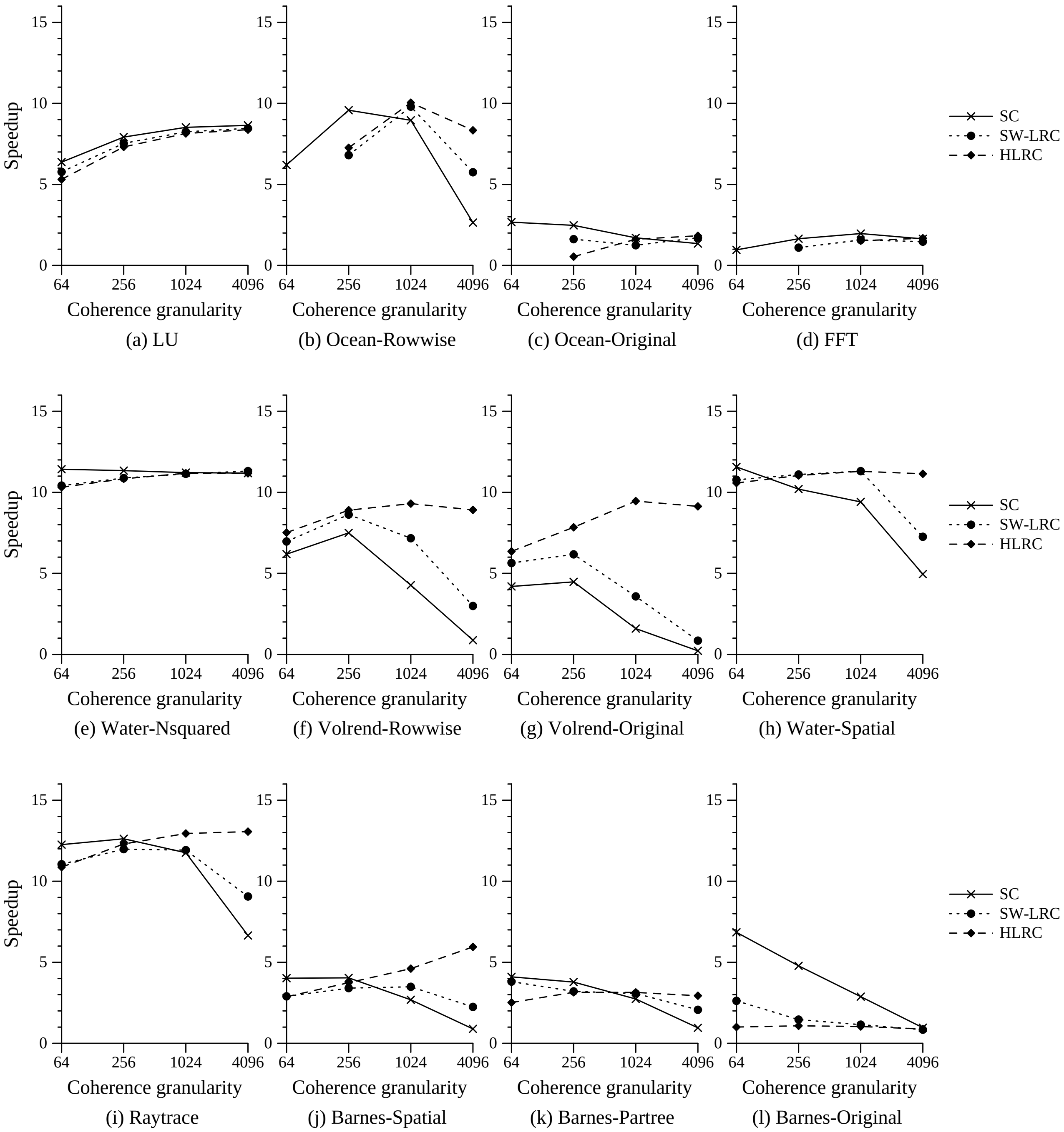

Figure 1: Speedups on T0 with 16 nodes. [Some numbers are missing because in those cases performance is dramatically affected by the disk swapping and becomes irrelevant for this study.]

- **Coarse-grain vs. Fine-grain data access** Data access granularity affects how the communication to computation ratio changes with the coherence granularity [12]. Applications with coarse-grain access tend to access a whole contiguous page at a time. Fine-grain applications are likely to scatter reads and writes across multiple pages. Fine-grain reads can introduce fragmentation with coarse coherence granularity and/or false sharing.

- **Coarse-grain vs. Fine-grain synchronization**

  The frequency of synchronization events is an important performance factor for release consistency protocols because all coherence events happen at synchronization. The frequency of synchronization can be reflected by the average computation time between two consecutive synchronization events. An application has fine-grain synchronization in a platform if the average computation time between two consecutive synchronization events is comparable to the cost for each synchronization event in that platform. For example, in our experiments, the minimum time in handling a synchronization event is around 150 microseconds, so if an application's average computation time between two consecutive synchronization events is less than several milliseconds, the application is classified as having fine-grain synchronization.

Table 2 summarizes the classification of sharing patterns and synchronization granularity in the applications.

5.2.2 Analysis in Categories

Now we analyze the application performance for each combination of false sharing pattern, access granularity, and synchronization style.

Single-writer with coarse-grain access

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	24654	6297	1574	393
	SW-LRC	24655	6297	1574	393
	HLRC	24655	6297	1574	393
Write Fault	SC	0	0	0	0
	SW-LRC	0	0	0	0
	HLRC	0	0	0	0

Table 3: LU

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	21803	6060	2593	3901
	SW-LRC		5128	1668	781
	HLRC		5176	1653	759
Write Fault	SC	4237	1232	392	187
	SW-LRC		1342	388	194
	HLRC		1269	368	176

Table 4: Ocean-Rowwise

LU is a typical application in this category, with good spatial locality and only one writer for each shared page. It is expected that all the protocols performs better at coarse granularity because of the effect of prefetching. The speedup curves and the frequency numbers validate the expectation.

Table 3 shows that the number of read misses decreases by about 4 times when the access granularity increases by a factor of 4 for all three protocols. No write fault occurs in all protocols because there is only one writer for each page. For the same reason, the HLRC protocol does not perform any diff operation. Furthermore, in LU, a page is never read by any nodes before it is written, so no invalidations are performed by the SC or LRC protocols.

Ocean-Rowwise also falls in this category. Although it is a single writer application with coarse grain access, but the problem size of 514×514 does not allow rows to be well aligned to pages, so fragmentation and false sharing exist at the boundary between two neighboring processors at page granularity. As shown in Table 4, write faults do occur and the number if write faults decreases as the granularity increases. Due to the fragmentation, the speedup drops at 4,096-byte granularity for all protocols. The HLRC protocol performs 50% better than the SW-LRC protocol, and 3 times better than the SC protocol because it greatly reduces the effects of the false sharing at the boundary.

Single-writer with fine-grain access

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	31714	28254	26707	11173
	SW-LRC		27310	25557	8886
	HLRC		27309	25608	8780
Write Fault	SC	1074	360	181	94
	SW-LRC		354	194	98
	HLRC		87	43	23

Table 5: Ocean-Original

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	92160	27360	11160	7110
	SW-LRC		27360	11160	7110
	HLRC		27360	11160	7110
Write Fault	SC	0	0	0	0
	SW-LRC		0	0	0
	HLRC		0	0	0

Table 6: FFT

Ocean-Original and FFT are example applications in this class. In Ocean-Original, writes are local but reads of border elements are remote. Reading elements at the column borders are fine grained. A contiguous allocation of partitions using four-dimensional arrays eliminates false-sharing ensuring a single writer for all pages which belong to the same block. Thus, there is little false sharing but significant fragmentation in the induced patterns. In FFT, the write access granularity is coarse while the read access granularity is fine-grained for this problem size [12] (a processor reads subrows of size 192 bytes from other processors).

All the protocols performs poorly. The best speedup is 2.7 for Ocean-Original and 1.9 for FFT. The main reason for this poor performance is the fragmentation due to the mismatch between the access granularity and communication granularity, which is more pronounced at a coarse granularity. For example, for Ocean-Original, with 4,096-byte block size, to read an 8-byte element in a column-oriented border, we need to fetch the full page containing that element. Therefore, with 4,096-byte block size, the unnecessary data traffic introduced by fragmentation is more than 99% of the total data traffic. With 64-byte block size, the fragmentation is still more than 88%.

For the SC protocol, fine granularity can improve the performance of the protocols by reducing the extra data traffic caused by fragmentation. So, for Ocean-Original, SC at 64-byte performs better than at coarse granularities.

Applications	Concurrent Writer per Block	Spatial Access Grnularity	Computation Time / Synch. (millisecs)	Number of Barriers	Temporal Synch. Granularity	Performance Level	Effect of Increasing Coherence Granularity
LU	single	coarse	71.69	64	coarse	all good	all improve
Ocean-Rowwise	single	coarse	9.88	323	coarse	all good	declines at 4K
Ocean-Original	single	fine	5.85	328	coarse	all poor	SC declines; LRCs improve
FFT	single	fine	170.36	10	coarse	all poor	inproves slightly
Water-Nsquared	multiple	coarse	59.93	12	fine	all good	all improve slightly
Volrend-Rowwise	multiple	fine	17.55	16	coarse	LRCs good SC medium	SC declines at 1K SW-LRC declines at 1K HLRC improves;
Volrend-Original	multiple	fine	17.55	16	coarse	HLRC good SW-LRC medium SC poor	SC declines at 1K SW-LRC declines at 1K HLRC improves;
Water-Spatial	multiple	fine	1439.83	18	coarse	all good	SC declines; SW-LRC declines at 4K HLRC stays constant
Raytrace	multiple	fine	100.87	1	coarse	all good	SC declines; SW-LRC declines; HLRC improves
Barnes-Spatial	multiple	fine	157.83	12	coarse	SC poor SW-LRC poor HLRC medium	SC declines at 1K SW-LRC declines at 4K HLRC improves
Barnes-Partree	multiple	fine	SC: 73.93 LRCs: 1.52	13	coarse	all poor	SC declines; SW-LRC declines; HLRC stays constant
Barnes-Original	multiple	fine	SC: 1.00 LRCs: 0.12	8	fine	SC medium; LRCs poor	SC declines; LRCs constant

Table 2: Classification and performance of sharing patterns and synchronization granularity in the applications.(Performance Level: good means speedups ≥ 8, medium means speedups ≥ 5, and poor means speedups < 5).

However, for FFT, fine granularity multiples the number of read misses due to lack of prefetching (see Table 6). Consequently, the overall performance for FFT decreases. For the SW-LRC and HLRC protocols, fine granularities increase the protocol overhead, so overall performance decreases for both Ocean-Original and FFT.

Multiple-writer with coarse-grain access

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	20487	14164	7821	1976
	SW-LRC	22059	13631	7074	1782
	HLRC	20489	12605	6553	1676
Write Fault	SC	8200	5130	2701	687
	SW-LRC	8791	5583	2670	852
	HLRC	8840	5521	2699	779

Table 7: Water-Nsquared

Water-Nsquared is an application with multiple writers per molecule. Its access pattern is migratory in the main phase of communication—updating the forces on the molecules. Since each process updates successively a large number of contiguous molecules, the migratory pattern at molecule level is preserved at page level which leads to a coarse-grain access pattern and large prefetching effects.

Table 7 shows that with 4,096-byte block size, the LRC protocols has fewer read misses than the SC protocol. This is due to the relaxed consistency. But the overhead of expensive protocol operations offsets that gain.

Multiple-writer with fine-grain access and coarse-grain synchronization

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	786	310	391	502
	SW-LRC	805	311	91	24
	HLRC	800	309	89	24
Write Fault	SC	45	18	16	6
	SW-LRC	50	34	31	21
	HLRC	33	24	26	16

Table 8: Volrend-Rowwise

Many irregular applications fall into this category. Of 12 applications, 5 are in this category: Volrend-Rowwise, Volrend-Original, Water-Spatial, Raytrace and Barnes-Spatial.

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	1430	812	895	2425
	SW-LRC	1294	529	143	43
	HLRC	1286	530	142	43
Write Fault	SC	345	388	505	496
	SW-LRC	385	467	635	575
	HLRC	134	105	40	23

Table 9: Volrend-Original

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	34340	51871	42086	39307
	SW-LRC	34243	20277	11485	4447
	HLRC	34238	20157	11350	4321
Write Fault	SC	1221	7041	4652	14288
	SW-LRC	1189	2349	1686	35832
	HLRC	19	332	325	436

Table 10: Water-Spatial

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	26027	12960	6700	6531
	SW-LRC	25456	11409	3566	1182
	HLRC	25400	11384	3603	1206
Write Fault	SC	1802	2607	4171	6490
	SW-LRC	1701	2558	4246	7313
	HLRC	146	240	414	403

Table 11: Raytrace

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	20098	14706	14160	17399
	SW-LRC	14346	7200	2567	751
	HLRC	14377	7259	2579	760
Write Fault	SC	5705	5329	5248	5449
	SW-LRC	6536	6557	6673	6445
	HLRC	4959	4162	1729	507

Table 12: Barnes-Spatial

All these applications exhibit both write-write and read-write false sharing at a coarse granularity of coherence. Their synchronization is coarse-grained because their average computation time between consecutive synchronization events is more than 17 milliseconds, 2 orders of magnitude more than the minimum time required to handle a synchronization event in our testbed. In Water-Spatial, the space cells containing the molecules are partitioned among processors, and a processor reads molecule data from its neighboring partitions. As the computation evolves, the molecules move from one cell to another, so the molecules in a processor's partition may fall on different pages, leading to a fine-grain access pattern. To a greater extent, in Barnes-Spatial, each processor accesses tree cells and particles that fall on different pages. Raytrace uses distributed task queues, and it updates pixels in the image plane as part of each task; both these access patterns are fine-grained and cause a lot of false sharing. Volrend-Rowwise and Volrend-Original are very similar to Raytrace in how they access task queues and the image plane. Volrend-Original has more false sharing than Volrend-Rowwise owing to the mismatch between the row-major memory allocation and the decomposition into small square tiles in the former.

These 5 applications illustrate the situations in which relaxed consistency protocols are needed to overcome the false sharing problem of large coherence granularities, since they perform much better than sequential consistency at coarse granularities. The HLRC protocol with 4,096-byte block size performs the best among all protocols and granularities, because the relaxed consistency model and the multiple-writer support reduce false sharing and the large communication granularity achieves some useful prefetching. Other than in Volrend-Original, Volrend-Rowwise and Barnes-Spatial, the SC protocol with 64-byte block size performs close to the best because the fine granularity nearly eliminates false sharing. Let us see why it doesn't perform comparably with HLRC at 4,096-byte granularity in these three cases. In Volrend-Original, the problem for SC is that write-write false sharing on the image is not eliminated even at 64-byte granularity, since the task size is made quite small (4×4 pixels) to achieve load balance (See Table 9). In Volrend-Rowwise and Barnes-Spatial, false sharing is not very significant at 64-byte granularity, but the loss of prefetching benefits makes the small granularity disadvantageous. Compared to the HLRC protocol with 4,096-byte granularity, the number of read misses for SC at 64-byte granularity is 24 times greater for Barnes-Spatial (Table 12), and 30 times greater for Volrend-Rowwise(Table 8).

Consider the different protocols at large granularities. SW-LRC performs better than SC because it alleviates read-write false sharing by delaying the invalidations. For example, the number of read misses in Water-Spatial under SW-LRC is only about 1/10 of that under SC (Table 10). Since HLRC is a multiple-writer protocol, it alleviates write-write false sharing as well. For example, it reduces the number of write misses from the SW-LRC and SC protocols by factors of 10 to 30 at coarse granularities (see Tables 8, 9, 10, 11, 12). As a result, it performs from 30% to 4 times better than the SW-LRC protocol with a 4,096-byte block size. As the block size decreases, the difference in number of write misses between both LRC protocols becomes smaller, so SW-LRC performance gets closer to HLRC performance.

Finally, at fine granularities such as 64 bytes, the high protocol overhead offsets any advantages of the relaxed software protocols. Since the relaxed protocols have more overhead, their performance is worse than the SC's performs by about 7 to 10%.

Multiple-writer with fine-grain access and fine-grain synchronization

Barnes-Original is an irregular application whose read-write accesses cause false sharing and fragmentation for almost all block sizes, especially in the tree-building phase

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	6932	9856	11642	14629
	SW-LRC	5280	5586	4451	3377
	HLRC	5184	5767	4388	3396
Write Fault	SC	1652	5223	5595	5645
	SW-LRC	2685	5484	5715	5853
	HLRC	2880	5277	4707	3853

Table 13: Barnes-Original

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Read Fault	SC	14152	13391	13036	16206
	SW-LRC	9498	7088	4566	1908
	HLRC	9773	8239	4690	1890
Write Fault	SC	4487	6267	5780	5757
	SW-LRC	5087	6947	6241	6189
	HLRC	5048	4472	3237	1701

Table 14: Barnes-Partree

Fault	Protocol	Coherence Granularity			
		64	256	1024	4096
Data Message	SC	8603	15154	17362	21091
	SW-LRC	7965	11071	10167	9231
	HLRC	10996	10920	9003	7252
Data Message Traffic	SC	0.53	3.70	16.96	82.39
	SW-LRC	0.49	2.70	9.93	36.04
	HLRC	0.46	1.63	4.52	13.76

Table 15: Barnes-Original data communication

that distinguishes it from Barnes-Spatial. In addition to the frequent locking of tree nodes in building the shared tree, the SW-LRC and HLRC protocols require adding synchronization to the application to make it comply with the release consistency model. Thus, Barnes-Original has substantially more synchronization events in the LRC versions than in the SC protocol. The application for the SC protocol issues a total of 2,086 lock calls at runtime, while the application for the LRC protocols issues 17,167 lock calls.

The average computation time between two consecutive synchronization events is 120 microseconds, even smaller than the minimum synchronization handling time. Barnes-Partree also requires adding synchronization to make it release consistent, but the number of locks in building the tree is much smaller since processors first build their local trees independently and then merge them. Although Barnes-Partree has many fewer locks than in Barnes-Original, the average computation time between consecutive synchronization events is still very small, only 1.5 milliseconds.

For Barnes-Original, using the relaxed protocols turns out to never be worthwhile even at 4096-byte granularity; they do not succeed in helping large granularities deliver the same performance as fine-grained SC. The reason, interestingly, is the high frequency of synchronization events. Although the effect of the relaxation is significant—4 times fewer read misses and 30% fewer write misses than with SC at the same granularity (see Table 13)—the resulting overhead reduction is negligible compared to the cost of the synchronization events in the relaxed protocols (note that synchronization events are much cheaper in SC since they do not involve protocol activity). The 15,179 additional lock operations account for more than 50% of the total execution time.

The other significant problem that remains at large granularities despite relaxed protocols is fragmentation. Table 15 shows that the data traffic under HLRC with a 4,096 block size is 25 times more than that under SC with a 64-byte block size. SW-LRC performs worse than HLRC at 4096 bytes because SW-LRC's data traffic is almost twice that of HLRC. As the block size decreases, these differences are reduced and SW-LRC becomes better than HLRC because of its lower protocol overhead.

By reducing synchronization, Barnes-Partree improves the situation for relaxed protocols significantly. With a 4,096-byte block size, HLRC is twice as good as SC, and 50% better than SW-LRC. But the alleviation of false sharing is still not enough to compensate for the high costs at the synchronization events: The HLRC protocol with 4,096-byte block size is 30% worse than the SC protocol with 64-byte block size.

5.3 Effect of Restructuring Applications

In our benchmark suite, there are several versions of Ocean, Volrend and Barnes. Their differences are in data structures, task partitioning, and algorithms for certain phases. Understanding the impact of these differences can give us some insights into the critical performance factors in making applications perform better on such systems.

As mentioned above, all protocols perform poorly for Ocean-Original because its fine grain access pattern causes large (88-99%) fragmentation. Ocean-Rowwise employs a rowwise partitioning strategy instead of the strategy of partitioning into square subblocks used in Ocean-Original. This change increases the inherent communication to computation ratio (which is a perimeter to area ratio), but it has the beneficial effect of moving the application from the class of single-writer applications with fine-grain access to that of single-writer applications with coarse-grain access. It therefore greatly reduces the total communication to computation ratio induced by large granularity, which dominates the inherent ratio. This change also simplifies the data structure. It no longer requires the use of complex, four-dimensional array data structures to keep partitions contiguous in the address space. It is therefore both easier to program than Ocean-Original and also significantly reduces the frequency of all overhead operations including faults, data traffic and control traffic. The best speedup over all protocols and granularities after the change increases from 2.7 to 10.0, achieved at 1024-byte granularity with HLRC.

Volrend-Original uses small square tiles of pixels as tasks, and is thus more load balanced in its initial partitioning than Volrend-Rowwise. However, the latter interacts much better with row-major memory layout, and has significantly less write-write false sharing at task borders. The HLRC protocol is less sensitive to false-sharing and more sensitive to the high overhead of synchronization needed for task stealing, so it performs slightly better with the better initial partitioning of Volrend-Original. However, the other two protocols are very sensitive to write-write false sharing and perform better with Volrend-Rowwise. In this case too, inherent algorithmic properties for which the original programs were generally optimized (like load balance and communication-to-computation ratio) are being traded off against system interactions (such as granularities and synchronization cost) to achieve better performance.

From the viewpoint of this study, the three different tree building algorithms in the three versions of Barnes differ mainly in the frequency of synchronizations and in their load balance characteristics. In the LRC protocols, the average computation time between consecutive synchronization events is 150 microseconds in Barnes-Original, 1.5 milliseconds in Barnes-Partree, and 157 milliseconds in Barnes-Spatial. These are averages over the entire application. The difference in frequency is in fact much worse since it is all concentrated in the tree-building phase. Barnes-Partree and Barnes-Spatial move us increasingly toward coarse-grained synchronization, though at the cost of increasing load imbalance in the tree building phase. With these protocols, the latter is a negligible problem compared to the benefits of the former. As a result, with 4,096-byte block size and the HLRC protocol, Barnes-Spatial performs 5 times better than Barnes-Original, and the gap with the other two protocols at 4,096-byte increases from 0 to a factor of 5. The tree building phase in Barnes-Spatial does not use locks, and processors are synchronized mainly by barriers. It reduces locking at the cost of some load imbalance, and therefore helps greatly when synchronization operations are expensive (as in HLRC) but can hurt when synchronization is less expensive and load imbalance takes on a greater role. For example, in SC at 64-byte granularity, more than 35% of the time in Barnes-Spatial is spent on barrier synchronization, and the performance is 40% worse than the same combination of protocol and granularity for Barnes-Original.

5.4 Interrupt vs. Polling

In the results we have presented so far, the polling method was used to service incoming messages. However, we have also evaluated the performance using interrupts. Our experimental results show that the polling method works better in most cases. However, none of our protocols perform consistently better with a single method for all granularities and all applications. Our results also show that

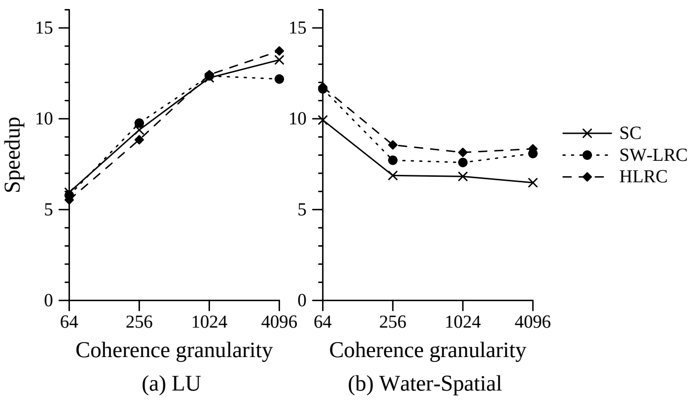

Figure 2: Speedups of LU and Water-Nsquared with interrupt mechanism on T0.

the SC protocol is more sensitive to which method is used than the SW-LRC and HLRC protocols. Due to lack of space, we only present speedup curves for two applications, LU and Water-Spatial(see Figure 2).

The interrupt method is straight forward. Message arrivals cause hardware interrupts, which are caught by the Solaris kernel and forwarded to the user process using Unix signals. Interrupts are disabled whenever we block for message arrival and while messages are being processed. Because Unix signals must cross between protection domains, the roundtrip time per notification is $\sim$ 70 microseconds. The polling method moves the overhead of crossing protection domains off of the critical path and runs entirely at user-level. It requires adding 7 instructions at each back edge in an application's control flow graph to check a control register for message arrival. Because the T0 device supports cachable control registers, the common case (that no message has arrived) incurs an overhead of only 6 or 7 cycles [28]. When a message does arrive, the round trip time for the mechanism is 1.5 microseconds, which includes the cost of clearing the T0 register with an uncached store.

The trade-off between the two mechanisms clearly depends upon the frequency of message arrivals: for frequent messages, polling works better, and vice versa. For coarse-grain applications like LU and Water-Nsquared, which send a small number of messages, interrupts perform significantly better across all protocols. For example, the LU application at the 4096-byte granularity performs 44-66% better with interrupts than polling, depending on the protocol. In fact, this application on one processor with the polling code inserted runs 55% slower than without the polling code.

A more interesting interaction between the message notification mechanism and protocols occurs with applications like Ocean-Rowwise, Raytrace, Volrend-Rowwise, Volrend-Original and Water-Spatial. In these applications, the SC protocol suffers from false sharing at large coherence granularities, resulting in the classic "ping-pong" effect as data moves between processors. In this case, the polling method performs poorly because blocks can be quickly stolen away; an invalidation request may be processed potentially as soon as the next backedge. Conversely, each time a block is obtained, the interrupt method temporarily disables interrupts to ensure forward progress. Because the timer resolution is 100 milliseconds, much greater than a typical basic-block size, the interrupt method delays the invalidations, allowing the processor to make multiple local accesses and reducing the rate of ping-ponging. The total number of misses under SC decreases significantly, down to 4%-70% of the polling case. In essence, the interrupt method approximates Dubois, et al.'s delayed consistency implementations, which are specifically targeted at reducing the impact of false sharing [8]. This problem is most marked with SC; however, SW-LRC also exhibits a milder form of the ping-pong effect, thus interrupts help reduce total traffic for it as well.

5.5 Discussions

Although the best way to understand performance is to look at applications individually, as we have done, and averages over parallel applications are not statistically significant, it may be useful after a detailed analysis to summarize with a broad brush (with all caveats about applications and platform noted). We play this game in this section. In particular, assuming that these are the only applications we care about, we ask questions about choices one might make among protocols and granularities and answer them using the data presented above.

For the first part of this discussion, for an application a, we only consider the original implementation used in the hardware cache coherence machine, ignoring the restructured implementations. That is, we only consider LU, Ocean-Original, FFT, Water-Nsquared, Volrend-Original, Water-Spatial, Raytrace and Barnes-Original.

For a given application a, for each combination of granularity g and protocol p, we define the quantity RE to be the relative efficiency of a protocol-granularity combination for a particular application. That is,

$$RE(a,p,g) = \frac{speedup(a,p,g)}{\mathrm{MAX(a)}}$$

where MAX(a) is the maximal speedup over all combinations of protocols and granularities for the given application a, and $speedup(a,p,g)$ is the speedup under the combination of protocol p and granularity g for that application.

We then define HM to be the harmonic mean of the relative efficiency (RE's) over 8 applications. For an application and a protocol, g_{best} is the granularity for which that protocol achieves the best speedup for that application. Similarly for an application and a granularity, p_{best} is the protocol which has the best speedup with that granularity for that benchmark. Table 16 presents all the HM values for all kinds of combinations including the ones with g_{best} and p_{best}.

Protocol	64	256	1024	4096	g_{best}
SC	0.746	0.791	0.522	0.134	0.872
SW-LRC	0.582	0.573	0.495	0.294	0.722
HLRC	0.446	0.426	0.546	0.517	0.571
p_{best}	0.803	0.878	0.798	0.540	1.000

Table 16: The HM values for all combinations across 8 applications only using the original implementation from SPLASH-2.

Now we can answer some questions we might want to ask in choosing among granularities and protocols. We see that given this testbed and the original implementations for this particular set of applications,

- If we have to choose a fixed protocol but are allowed to choose granularities on a per-application basis, SC is the best among the three protocols. The HM value under the combination of the best granularity and the SC protocol is 0.872, higher than the combinations of the best granularity with the other two protocols.
- If we have to fix the granularity but are allowed to choose protocols on a per-application basis, 256-byte is better than the other three granularities. The HM value under the combination of 256-byte granularity and the best protocol is 0.878.
- If we have to fix both the granularity and the protocol, the combinations of the SC protocol and 256-byte granularity are better than the other combinations. The HM values under this combination is 0.791.

The results for the combinations with coarse granularities are hurt significantly because of the original Barnes application.

Protocol	64	256	1024	4096	g_{best}
SC	0.753	0.867	0.717	0.274	0.955
SW-LRC	0.400	0.749	0.823	0.558	0.861
HLRC	0.388	0.758	0.903	0.927	0.956
p_{best}	0.773	0.895	0.935	0.930	1.000

Table 17: The HM values for all combinations across 8 applications using all impelementations for each application.

In the statistics presented so far we considered the original implementation for each application. However, we have seen that the performance of some applications for all protocols can benefit greatly from restructuring. In what follows we compute the same statistics and answer the same questions by choosing, for each combination of protocol and granularity, the implementation or version of the application which delivers the best performance under this combination [2]. So we change the definition of the relative efficiency(RE) to

$$RE(a,p,g) = \frac{Max(a,p,g)}{\mathrm{MAX(a)}}$$

where MAX(a) is the maximal speedup over all versions for a given application a, and $MAX(a,p,g)$ is the best speedup for protocol p and granularity g among all versions of that application. Table 17 presents all the HM values using this method for all kinds of combinations choosing the best implementation for each combination of protocol and granularity. The answers to the questions now are:

- If we have to choose a fixed protocol but are allowed to choose granularities on a per-application basis, either the SC or the HLRC protocol is the best choice on average. The HM value for the best granularity and the SC protocol is 0.955, while it is 0.956 for the HLRC protocol.
- If we have to fix the granularity but are allowed to choose protocols on a per-application basis, any relative large granularities(256-byte, 1,024-byte and 4,096-byte) are good. The HM values for the combinations of best protocol and these three granularities are respectively 0.895, 0.935 and 0.930, compared to 0.773 of the best protocol at 64-byte.
- If we have to fix both the granularity and the protocol, the combination of the HLRC protocol and 4,096-byte granularity seems to be the best choice. The HM value under this combination is 0.927.

Both larger granularities and more relaxed protocols seem to become more attractive when the improved versions of the applications are included in the mix, which is not very surprising since the improvements, while general, were largely designed to interact better with these situations. It will be interesting to see if the performance of SC at larger granularity can be further improved by alternate restructurings.

[2] Water-spatial and Water-Nsquared are always treated as different applications, since they use completely different algorithms and may produce different results, so we are always averaging over 8 applications in computing these statistics.

6 Related Work

There is a large body of literature in the area of distributed shared memory. The most related work to this paper includes research on relaxing consistency models and providing fine-grained coherence granularity for software coherent shared memory.

The original shared virtual memory (SVM) proposal and prototype [20] uses the traditional virtual memory access protection mechanism to detect access misses and implements the sequential consistency model [17] on a network of workstations. The coherence unit of the prototype is a 1,024-byte virtual memory page.

Since then, two main approaches have been taken to deal with the false-sharing and fragmentation problem in SVM systems: relaxing consistency models and providing fine-grained access control. Examples of relaxed consistency models and systems include release consistency (RC) [10] and its SVM implementation [5], delayed consistency model [8] and its SVM protocols [4, 9], multiple-writer lazy release consistency (LRC) model [15] and implementation [14], entry consistency model and prototype [2], automatic update release consistency [11], scope consistency [13], home-based lazy release consistency and its implementations [30], and single-writer lazy release consistency [16]. All prototypes based on relaxed consistency models use virtual memory page sizes as their coherence units.

Another approach is to preserve the sequential consistency model and to find ways to reduce the coherence granularity. Examples of providing fine-grained access control include taking advantage architectural features such as the ECC bits to trap access faults [27], using software instrumentation for shared reads and writes [27, 26], and building special access control hardware for commodity workstations [23].

Keleher [16] compares sequential consistency with single writer and multiple-writer LRC protocols and concludes that overall, the multiple-writer version is only 9% better than the single-writer one and 34% better than the sequential consistency one. Our study compares the three protocols for different granularities and for a larger class of applications including 7 irregular applications out of 12.

Although previous implementation or simulation studies have compared various protocols with various kinds of applications, none has studied the performance tradeoffs between relaxing consistency models and providing fine coherence granularity.

7 Conclusions

We have investigated the performance tradeoffs of relaxed consistency models, coherence granularity, and mechanisms for handling message arrivals on a cluster of workstations with hardware access control but software coherence protocols. Our results can be summarized as follows.

No single combination performs of protocol and granularity performs best for all applications. Two combinations generally perform well. The SC protocol with fine granularity works well with 7 applications. The combination of a home-based lazy release consistency (HLRC) protocol and 4,096-byte granularity works well with 8 applications. Barnes-Original performs substantially better with the combination SC-64 than with HLRC-4096, whereas Volrend-Rowwise and Volrend-Original perform substantially better with the combination HLRC-4096 than with SC-64.

The best granularity for the HLRC protocol is 4,096 bytes for almost all applications, whereas for the SC protocol the best granularity is usually between 64 and 256 bytes. The best granularity for the SW-LRC protocol varies among applications. Similarly, at 64 byte granularity SC is almost always the best protocol, while at 4,096 byte granularity HLRC is always best.

In most cases in which SC experiences performance losses when granularity is increased, the performance is regained (and sometimes improved significantly) by using relaxed protocols, particularly HLRC. Barnes-Original is the notable exception. The major circumstance in which HLRC does not work well is when synchronization frequency is high, either inherently in the algorithm or to make the program release consistent.

For most of our irregular applications, the multiple-writer HLRC protocol performs substantially better than single-writer SW-LRC protocol with coarse granularities. With 4,096-byte granularity, HLRC performs better than SW-LRC in all applications. The performance difference is large in some cases: factors of 3 and 10 respectively for Volrend-Rowwise and Volrend-Original.

Finally, we find the polling method for service incoming messages works better in most cases. However, none of our protocols perform consistently better with a single method for all granularities and all applications. Our results also show that the SC protocol is more sensitive to which method is used than the SW-LRC and HLRC protocols.

Overall, for our applications and platform, when we use the original versions of the applications ported directly from hardware-coherent shared memory, we find that the SC protocol with 256-byte granularity performs best on average. However, when the best versions of the applications are compared, the balance shifts in favor of HLRC at page granularity.

Our study has several limitations. We have not studied block sizes greater than 4,096 bytes, and have not been able to run large problem sizes due to memory limitations. We have also not examined delayed consistency protocols that can delay invalidation messages to some extent without using high-overhead protocol operations at synchronization points. We have not examined the memory utilization of different protocol and granularity combinations. Finally, this study has not examined all-software systems, since access-control was performed in hardware on the platform we used. It would be interesting to also investigate all-software systems that provide fine-grained access control through software instrumentation of loads and stores [27, 26] or page-grained access control entirely through the virtual memory mechanism.

References

[1] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 125–134, May 1990.

[2] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93 Conference*, February 1993.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[4] L. Borrmann and M. Herdieckerhoff. A Coherency Model for Virtual Shared Memory. In *Proceedings of the 10th International Parallel Processing Symposium*, June 1990.

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

[6] David Culler, Lok Tin Liu, Richard Martin, and Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, pages 35–43, February 1996.

[7] David Culler, Lok Tin Liu, Richard Martin, and Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, pages 35–43, February 1996.

[8] M. Dubois, J.C. Wang, L.A. Barroso, K. Lee, and Y-S Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. In *Supercomputing '91*, pages 197–206, 1991.

[9] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[11] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[12] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[13] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[14] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[15] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[16] P.J. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the IEEE COMPCON '96 Conference*, February 1996.

[17] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[18] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[19] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume II Software, pages 94–101, August 1988.

[20] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[21] Scott Pakin, Mario Laura, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[22] Robert W. Pfile. Typhoon-Zero Implementation: The Vortex Module. Technical report, Wisconsin University, CS department, 1995.

[23] S. K. Reinhard, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[24] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 325–336, April 1994.

[25] ROSS Technology, Inc. *SPARC RISC User's Guide: hyperSPARC Edition*, September 1993.

[26] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, SOftware-Only Approach for Supporting Fine-Grain Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[27] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood. Fine-grain Access for Distributed Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[28] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, March 1996.

[29] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *The Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[30] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.