Home-based SVM protocols for SMP clusters:
Design and Performance

Rudrajit Samanta[1], Angelos Bilas[1], Liviu Iftode[2], and
Jaswinder Pal Singh[1]

[1] Department of Computer Science, Princeton University
Princeton, NJ 08544
[2] Department of Computer Science, Rutgers University
Piscataway, NJ 08855

{rudro, bilas, jps}@cs.princeton.edu, iftode@cs.rutgers.edu

Abstract

As small-scale shared memory multiprocessors proliferate in the market, it is very attractive to construct large-scale systems by connecting smaller multiprocessors together in software using efficient commodity network interfaces and networks. Using a shared virtual memory (SVM) layer for this purpose preserves the attractive shared memory programming abstraction across nodes. In this paper:

- We describe home-based SVM protocols that support symmetric multiprocessor (SMP) nodes, taking advantage of the intra-node hardware cache coherence and synchronization mechanisms. Our protocols take no special advantage of the network interface and network except as a fast communication link, and as such are very portable. We present the key design tradeoffs, discuss our choices, and describe key data structures that enable us to implement these choices quite simply.

- We present an implementation on a network of 4-way Intel PentiumPro SMPs interconnected with Myrinet, and provide performance results.

- We explore the advantages of SMP nodes over uniprocessor nodes with this protocol, as well as other performance tradeoffs, through both real implementation and simulation as appropriate, since both have important roles to play.

We find one approach to deliver good parallel performance on many real applications (at least at the scale we examine) and to improve performance over SVM across uniprocessor nodes.

1   Introduction

Small-scale shared-memory multiprocessors are becoming increasingly widespread and clusters of these machines are becoming popular computing platforms. Given this development and the attractiveness of the shared address space programming model, it is natural to examine whether the page-based software shared virtual memory (SVM) approach developed for uniprocessor nodes extends well to using cache-coherent multiprocessors as the basic nodes, and how the protocols can be improved for this task. In this environment, the hardware coherence protocol operates at cache line level within a node, and the higher-overhead SVM mechanism provides communication and coherence at page granularity across nodes. Another view of this approach is that the less efficient SVM is used not as the basic mechanism with which to build multiprocessors out of uniprocessors, but as a mechanism to extend available smaller-scale machines to build larger machines while preserving the same desirable programming abstraction. The trends in the marketplace make this two-level approach attractive. A network of bus-based, symmetric multiprocessors (SMPs) forms our own next-generation SVM infrastructure, and we focus on SMP nodes in this paper.

The best known all-software SVM system (across uniprocessors) is TreadMarks from Rice University [15]. Recently, however, all-software versions of multiple-writer home based protocols - which were originally developed for systems with hardware support for automatic write propagation [3, 10] - also have been developed. The idea, proposed in [11], is to compute diffs as in previous all-software protocols, but to propagate the diffs to the home at a release point and apply them eagerly, keeping the home up to date. Page faults are satisfied by obtaining the page from the home, where it is guaranteed to be up to date according to release consistency. In a preliminary evaluation on the Intel Paragon multiprocessor [25], this software home-based protocol (called *HLRC*) was found to outperform earlier distributed all-software protocols, just as home-based protocols with hardware write propagation [3, 17, 9] were found to earlier, and is thus attractive.

A simple way to build an SVM system across SMPs is to treat each processor as a separate node in a uniprocessor-node SVM protocol, using the SMP hardware simply to accelerate message passing [5]. Other previous implementations have examined single-writer protocols based on directory approaches [8]. The results from the latter implementation were not very optimistic. A multiple-writer directory based scheme was implemented concurrently with our work, and takes great advantage of the features of the DEC Memory Channel network interface and network [22]. There has also been an SMP-node implementation of a fine-grained approach to software shared memory [18]. Our goal is to

build and evaluate a multiple-writer home-based lazy release consistency protocol, *HLRC-SMP*, that takes advantage of hardware coherence and synchronization within a node as far as is advantageous, yet preserves laziness even within a node so that a process's pages are not invalidated unnecessarily, and does not take special advantage of network interface features.

We first discuss several design choices, together with the choices we made and their relative merits in Section 2. We describe our resulting SVM protocol and the key data structures that enable us to implement our choices quite cleanly. After briefly introducing the applications we use, in Section 3, we evaluate our actual implementation on PentiumPro Quad SMPs connected by Myrinet in Section 4. Performance data and the associated analysis is presented here.

In Section 5 we examine the tradeoff between using uniprocessor and SMP nodes as the building blocks of an SVM system for a fixed total number of processors. While SMP nodes contain some communication and synchronization within the node, they often increase the bandwidth demands placed on the node-to-network interface since the interface is shared by multiple processors. We analyze whether SMP nodes with our protocol improve performance significantly over using uniprocessor nodes, and how the results vary with the size of the SMP and the bandwidths of the node to network interfaces. The available platform for our real implementation is too small to compare SMP with uniprocessor nodes (4, 4-processor SMPs), and is inflexible in the performance parameters and clustering configurations it allows. It also suffers from some OS artifacts, making comparisons difficult. We therefore perform this comparison through very detailed simulation.

We find that out of ten regular and irregular applications, performance improves substantially with the use of four-processor SMPs in six of them; in three there is a much smaller improvement, and in one performance declines unless node-to-network bandwidth is increased substantially with SMP size. Increasing node-to-network bandwidth with SMP size almost always improves performance somewhat, except in one case when the key bottleneck going to even bigger (8-way) nodes turns out to be contention on the SMP memory bus.

2 Protocols

The SMP protocol we present is based on home-based lazy release consistency (*HLRC*) [25]. In this section we briefly review *HLRC*, discuss the design choices for the protocol extensions to support SMP nodes, and present the specific choices made in our implementation.

2.1 Home-based LRC Protocol

Home-based Lazy Release Consistency (*HLRC*) [25] is a variation of *LRC* [16] which uses a software write detection and diff-based write propagation scheme. *HLRC* uses a "home" node for each shared page at which updates from writers of that page are collected. This protocol is inspired by the design of Automatic Update Release Consistency (AURC) [10]. The difference is that updates are detected in software and propagated using diffs unlike in AURC, so *HLRC* does not require any special hardware support.

In *HLRC* diffs are computed at the end of each time interval (for instance, during a release operation) for all pages that were updated in that interval. Once created, diffs are eagerly sent to the home nodes of the pages where they are immediately applied. Writers can discard their diffs as soon as they are dispatched. Home nodes apply arriving diffs to the relevant pages as soon as they arrive, and immediately discard them. A vector time-stamp approach is used to preserve the lazy release consistency model [16]. During a page fault following a coherence invalidation (performed at acquires), the faulting node fetches the up-to-date version of a whole page from the home node.

HLRC has some advantages when compared to standard *LRC*: (i) accesses to pages on their home nodes cause no page faults, (ii) diffs do not need to be created when the writer is the home node, (iii) non-home nodes can always bring their shared pages up-to-date with a single round-trip message, and (iv) protocol data and control messages are much smaller than under standard *LRC*. the disadvantages are that whole pages are fetched and good home assignment may be important. An implementation of *HLRC* on the Intel Paragon has showed that *HLRC* outperforms a traditional *LRC* implementations, at least on the platform and applications tested, and also incurs much smaller memory overhead [25]. *HLRC* achieves ordering of all messages completely in software.

2.2 Home-based Protocol LRC for SMP nodes

2.2.1 High-level Design Alternatives

There are three major alternatives for managing the "processes" and their view of the data within an SMP node. One extreme for each process have it's own separate physical address space and page table and not share any state with the other processes in the SMP through the hardware coherence mechanism. All processes, within or across SMP nodes, communicate and synchronize via the SVM software protocol, and the hardware within an SMP is used only to accelerate the underlying message passing used by the SVM library. The other extreme is to use threads within an SMP. The threads on and SMP share a physical address space and the page table. Hardware sharing and synchronization are used within an SMP, but the disadvantage is that all threads in an SMP have the same view of the data : when one thread decides to invalidate a page according to the consistency model, that page is automatically ("eagerly") invalidated for the other threads as well. The required global TLB flushes for each page invalidation can prove to be expensive in this scheme. However, since all threads within the same SMP have the same state for a given page, local acquire operations are very cheap. The first alternative has been shown to perform poorly in [22] due to high overheads.

HLRC-SMP is used as an intermediate model that achieves both hardware sharing and laziness within the SMP node. Hardware sharing and (most) synchronization is achieved by having the processes in an SMP share a physical address space. Laziness is achieved by allowing the processes to have separate page tables so they can each have their own view of the state of a given page. The challenge is to support laziness within a node with little software overhead for intra-node activity. Our simulation results show that the the threads model, leads to more page faults and invalidations in some irregular applications. For example, Barnes exhibits a 10% greater number of page faults in the thread model when compared to the process model.

Another major design issue is how protocol processing

should be performed for incoming remote requests. One approach is to dedicate one or more processors in the SMP solely to protocol processing, having them poll for incoming requests. However, protocol activity is usually small and this approach is very wasteful of computational resources. Alternatively, in SVM systems where code instrumentation of compute processes for polling is not used, interrupts are delivered to some process at protocol requests. The alternatives include using a separate process (which can run on any compute processor) for protocol processing or having one of the compute processes perform all protocol processing, along with useful their computation. *HLRC-SMP* uses the first approach, since it allows more opportunities for load balancing the protocol processing across processors and because it leads to a cleaner implementation. Finally, *HLRC-SMP* implements hierarchical barriers and TLB invalidations are avoided with a scheme very similar to the 2-way-diffing described in [22].

2.2.2 Implementation

This section presents the data structures that allow us to easily implement the above choices in the *HLRC-SMP*, and how the synchronization and page fault operations are managed using them.

2.2.3 Data structures

To illustrate the data structures, we first need to define some key terms. The time during the execution of a parallel program is broken into *intervals*. With uniprocessor nodes, intervals are maintained on a per-process basis and numbered in a monotonically increasing sequence. An interval is the time between two consecutive synchronization operations by a single process. In *HLRC-SMP*, intervals are maintained on a per-node, and not a per-process basis. The interval number is incremented when *any* process within the SMP performs a synchronization operation. Each process maintains a list called the *update-list*, which records all the pages that have been modified by this process in the current interval. When any synchronization operation happens, the process performing this operation ends an interval for the node.
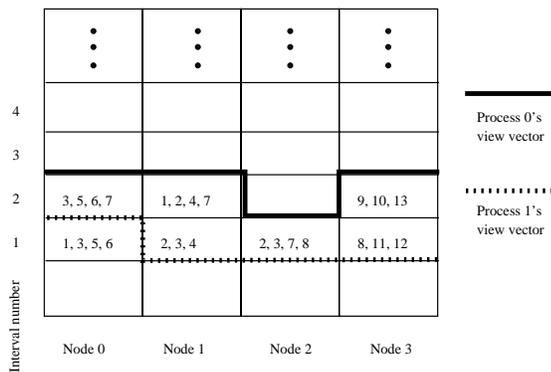

Figure 1: The bins data structure

When a process ends its interval, it's update-list is placed in a data structure called the *bins* (see Figure 1). Once a process "grabs" a bin, no other process will be able to write to the same bin. This is the key data structure used by the

SMP protocol. In our SMP protocol we use a single column for each node and not for each process. Thus, the bins data structure scales with the number of nodes and not the number of processors in the system, providing better scalability than schemes whose data structures are proportional in size to the number of processors. This would be even more important if the nodes were large, e.g. if we were using SVM to connect several medium-scale DSM machines.

Each SMP node contains a copy of the entire bins data structure. The entry in a given bin (say interval 2, node 1) will be the same in every node's copy of the bins data structure. However, for a particular node A, when the entry for another node B for a particular interval will appear in A's bins depends on when A does a causally related acquire from B and receives them. Essentially the bins inform us which pages were modified in relation to which synchronization operation, and in which order, thus telling us which pages need to be invalidated during a synchronization operation to maintain consistency. This information is also used to infer version information about the pages, so that when we fetch pages from the home node, we know which versions to demand.

So far the bins data structure does not provide different processes within an SMP with different views of the world. To provide laziness within a node, only a simple per-process data structure is required. We refer to this as the *view vector*. Essentially this is the "view of the world" that a process has. This vector maintains the information regarding what portion or height of the bins for different nodes has been "seen" (i.e. the invalidations corresponding to those intervals from different nodes have been performed) by this particular process. When one process fetches new bin information from another node, on an acquire the new information is available to all processes in it's SMP node. However, this and the other processes do not *act* on all this information immediately. Processes only invalidate pages for their own acquires, and only the pages (i.e. the entries in the bins) that they are required to "see" (as dictated by the last releaser of the lock).

2.3 Operations

Let us see how the bins and the view vectors work in conjunction with various synchronization operations.
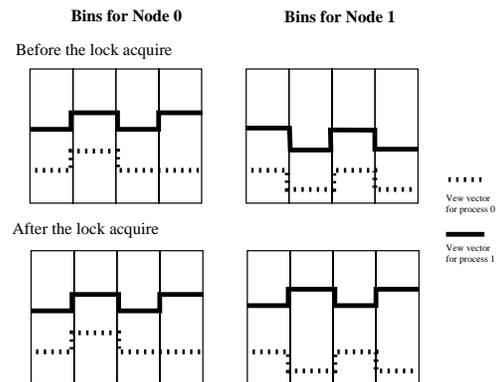

Figure 2: A remote lock acquire

Lock acquires: During a remote lock acquire operation (i.e. when the requested lock is available at a remote SMP node), the requester sends over a vector which indicates what bins are currently present at this node (i.e. a vector that specifies the height of it's bins). Next, any portion of the bins that are not available at the requester but exist at the releaser are sent back in the form of *write notices*, along with the view vector of the process that released the lock (as it was when the lock was released). Finally, the requesting process matches it's view vector with the lock releaser's view vector, by invalidating, for itself only, all the pages indicated in the bins that are "seen" by the releaser's view vector but not by the requester. This operation is illustrated in Figure 2. In this example process 1 on node 1 is acquiring a lock which was previously released by process 1 on node 0.
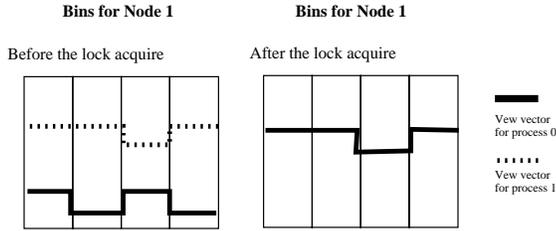

Figure 3: A local lock acquire

A local lock acquire operation (i.e. the acquirer and releaser reside on the same SMP) is completely local with no external or internal messages. All that is required is the matching of the requesting process' view vector with the releaser's, and the necessary invalidations, so there is only a small amount of protocol processing. As shown in Figure 3, this operation only involves bins available at this node and no data transfers across nodes are involved.

Barriers: For a barrier operation, the processes within a node barrier first and update the bins. Then, all the bins that have been generated at this node and have not been propagated yet are sent to the barrier manager. The barrier manager then disperses this information to all the nodes (not processors) in the system. Now all the bins at all nodes are filled to the same height. At this time all processes in each node match their view vectors to one that includes all the available bins, invalidating all the necessary pages.

Local copy updating: Finally, let us see how data is fetched when a more recent version of a page is needed. The home node of a page always has the most current version of the page. However, updates (in the form of diffs) may be in flight, so it is desirable when requesting a page to specify the version that is absolutely necessary. To achieve this we use a system of *lock time-stamps* and *flush time-stamps*. Each page at the home has associated with it a flush time-stamp that indicates what is the latest interval for each node for which the updates are currently available at the home. One may think of this vector as a "version" of the page. On a page fault, the requester sends to the home a lock time-stamp, corresponding to it's last lock acquire, which indicates to the home what the flush time-stamp of the page should at least be, in order to ensure that all relevant changes to the page by other processors are in place. If the flush time-stamp is less up to date then the lock time-

stamp, then diffs must be on their way so the home waits for the diffs to arrive before satisfying the page request.

During the page fetch operation, there is one major complication. If there are outstanding updates to a page (which have not been communicated to the home via diffs) or if other processes in the SMP are actively writing to the page in question, fetching the page in its entirety will destroy these updates to the page. To avoid this problem we use a "dirty bit" which detects this scenario. This bit is set on the first write by any process in the SMP to that page. It is cleared when a diff is performed on the page and there are currently no active writers to the page. When a page is observed to be "dirty" we perform a *safe page fetch*. Essentially a safe page fetch diffs the page returned by the home with the twin of the page and applies these updates to the current local page as well as the current twin for this page. This scheme is similar to the "2 way diffing" used in Cashmere-2L [22].

3 Applications

We use several applications from the SPLASH-2 [23] application suite to evaluate the *HLRC-SMP* protocol. We now briefly describe the basic characteristics of each application that are relevant to the use of SMP nodes. A more detailed classification and description of the application behavior for SVM systems with uniprocessor nodes is presented in [12]. The applications can be divided in two groups, regular and irregular.

The regular applications are FFT, LU and Ocean. Their common characteristic is that they are single-writer applications a given word of data is written only by the processor to which it is assigned. Given appropriate data structures they are single-writer at page granularity as well and pages can be allocated among nodes such that writes to shared data are mostly local. There is no need to compute diffs for these applications. Protocol action is required only to fetch pages. The applications have different inherent and induced (at page granularity) communication patterns [23, 12], which affect their performance and the impact on SMP nodes.

FFT: The all-to-all, read-based communication in FFT is essentially a transposition of a matrix of complex numbers.

LU: The contiguous version of LU is single-writer and exhibits a very small communication to computation ratio, but it is inherently computationally imbalanced.

Ocean: The communication pattern is largely nearest-neighbor and iterative on a regular grid. We run the contiguous (4-d array) version of Ocean as well as one that partitions the grid into blocks of rows rather than square subgrids [13].

The irregular applications in our suite are Barnes, Radix, Raytrace, Volrend, Water-nsquared and Water-spatial.

Barnes: Access patterns are irregular and fine-grained. We use two versions of Barnes. The first version (Barnes-rebuild) uses the tree-building algorithm in SPLASH-2. The second version (Barnes-spatial) [13] is optimized for SVM. It avoids locking entirely during tree building at the cost of load imbalance in that phase.

Radix: The radix sorting application exhibits highly scattered and irregular remote writes.

Raytrace: In the ray tracing application, a global lock in the SPLASH-2 version that was not necessary was removed, and task queues are implemented better for SVM and SMPs. Inherent communication is small.

Volrend: Compared to the SPLASH-2 version, a better balanced initial assignment of tasks to processes is provided, since task stealing is very expensive in SVM.

Water-nsquared: Water-nsquared can be categorized as a regular application, but we put it here to ease the comparison with Water-spatial. Updates to molecules due to force calculation are accumulated locally between iterations and performed to remote data only once at the end of each iteration.

Water-spatial: This exhibits irregular but near-neighbor oriented communication.

4 Prototype implementation

The nodes we used to implement *HLRC-SMP* are Intel SMPs each with 4 PentiumPro CPUs running at 200MHz. The PentiumPro processor has an 8KB Data and an 8KB Instruction cache. The processors are equipped with a 512KB unified L2 cache and there is 256MB of main memory per SMP.

There are currently 4 SMP nodes connected by a Myrinet network. Each network interface has a programmable chip and connects the node to the network with two unidirectional links of 160MB/s peak bandwidth each. Actual node-to-network bandwidth is usually constrained by the 132MB/s I/O bus. All nodes are connected directly to an 8-way switch.

4.1 Basic Software and Costs

For basic communication we use the Virtual Memory Mapped Communication (VMMC-2) library for the Myrinet network [6]. This general-purpose, memory-mapped communication layer provides a minimum one-way end-to-end latency of approximately 10 μs and a peak bandwidth of about 100MB/s on this hardware.

The SMP nodes run the SMP version of the Linux operating system (version 2.0.24). There are two slight modifications made to the Linux kernel to provide better performance for certain operations, discussed later.

The costs of various basic operations in our environment are shown in Table 1 [1].

Basic Operation	Cost (μs)	Micro-benchmark	Cost (μs)
mprotect	12	1way latency (1 word)	10
Read fault (null)	20	Page transfer b/w	50MB/s
Write fault (null)	20	Page fetch	200
bcopy (4KB)	150	Remote Lock Acquire	220
		Barrier (16 procs)	500

Table 1: Costs on the implementation platform. All costs are in μs, excpept if otherwise noted.

The cost of the page-fetch operation is much faster than previous implementations of SVM systems, for instance TreadMarks [16] reports over 1900 μs (for a 4KB page) and SoftFlash [8] states 1164μs (for a 16KB page at the Kernel level on an SGI Challenge). Cashmere-2L [22] reports the page fetch operation to cost about 800 μs. Our system benefits from the very low latency of the VMMC-2 implementation and of Myrinet.

---

[1]The *bcopy* of a single page results in a bandwidth of 26 MB/s. The Lock Acquire and the Barrier operations do not include the cost of exchanging coherency information (invalidations).

4.2 Performance

In this section we present some preliminary performance results evaluating the *HLRC-SMP* implementation on this 4×4 platform. We present results for a number of Splash-2. Speedups are reported in Table 2 along with the problem sizes. Figures 4-5 present the implementation breakdowns for 8 applications. The performance of the applications seem to fall into 3 major categories: good, medium and bad. For the applications that fall into the "good" category, there is really not much need for a detailed exploration since the breakdowns speak for themselves.

Application	Problem size	Uniprocessor time (s)	Speedup
FFT	1M points	4.59	1.27
LU (contiguous)	2048 x 2048	133.80	9.63
Ocean (rowwise)	514 x 514	17.98	3.57
Barnes (rebuild)	16384 bodies	23.26	2.09
Barnes (spatial)	16384 bodies	22.94	10.27
Raytrace	car	30.00	10.40
Volrend	head, 4 frames	10.71	10.95
Water (nsquared)	4096, 4 steps	276.50	11.32
Water (spatial)	4096, 4 steps	48.29	7.34

Table 2: Speedups for 16 processors (4 nodes)

The 2 applications in the "medium" category are LU contiguous, Ocean rowwise and Water spatial. The barrier costs in LU are due to load imbalance, and in Water require reasonable fraction of the time. The problem in the latter appears to be the Data wait time. Our investigations show that a very high page fetch cost is the major reason for this. By running the application with smaller numbers of processes per node and looking at the average page fetch time, we observe that contention for protocol handling is a major contributor to this time. The other cause related to this is the time required for the interrupt on message arrival to be delivered to the user application: during this operation, process scheduling on the SMP systems also presents a problem and adds to our costs. Ocean-rowwise shows a higher than expected computation time, which results in a lower speedup.

Finally, the applications for the "bad" category are Barnes-rebuild and FFT. These applications are known to be sore points for most SVM implementations [12]. Barnes-rebuild uses a huge number of locks and has a fine grained data access pattern, both of which are weak points of most page-based SVM systems. FFT has extremely bursty communication at barriers and places a serious strain on network bandwidth and latency. The scattered, fine-grain and bursty communication pattern for Radix is the primary reason for its poor performance.

4.3 Implementation Issues

We discuss a few key implementation issues that we encountered and found to be important. The solutions here were incorporated in the results resented above.

Performing diffs: Performing diffs does occupy a nontrivial fraction of the time spent in the SVM system. To avoid these unnecessary diffing, we implement a strategy where we diff only at a barrier and when a lock is sent out to a remote node. This lazy diffing scheme helps to significantly reduce the cost of local operations (especially lock releases). However, the scheme does introduce a certain amount of
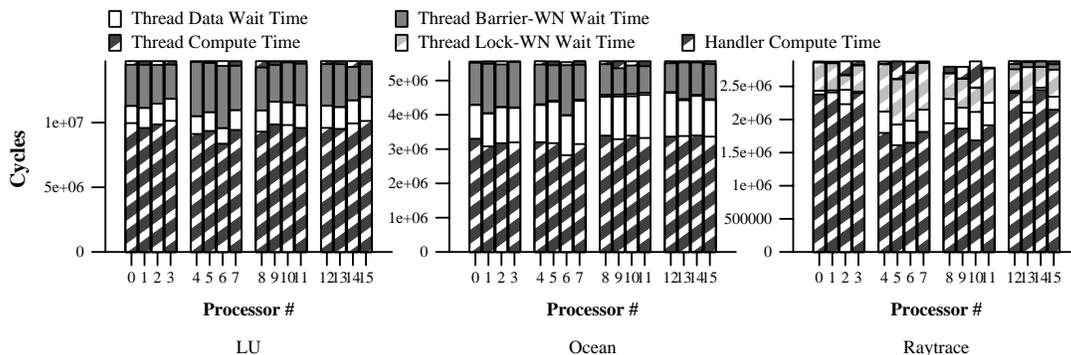
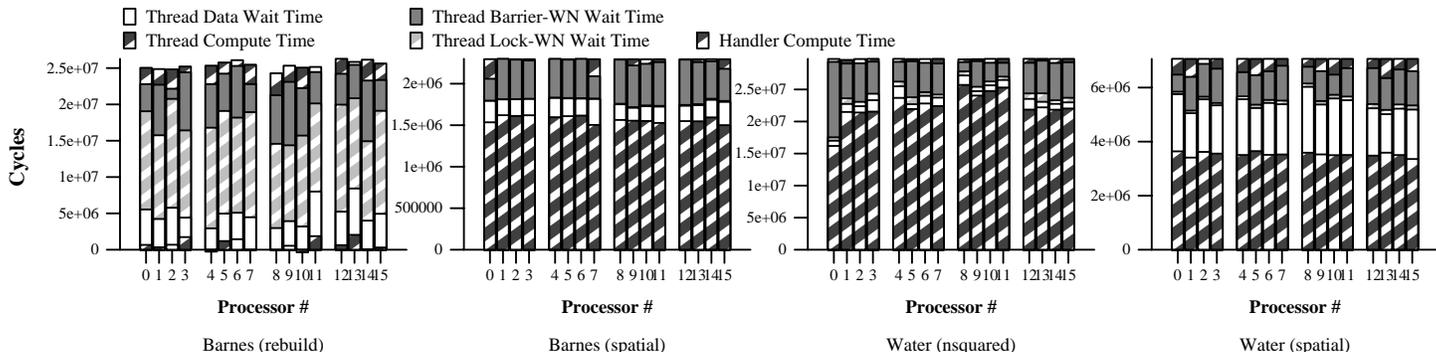Figure 4: HLRCsmp breakdowns for LU (contiguous), Ocean (rowwise) and Raytrace.

Figure 5: HLRCsmp breakdowns for Barnes (rebuild), Barnes (spatial), Water (nsquared) and Water (spatial).

complexity. Basically, attempting to fetch a page for which we have outstanding updates waiting to be diffed will result in a loss of these updates. This is avoided by marking the page as "dirty" which will cause all fetches of that page from the home to be safe page fetches as described in Section 2.3.

Invalidation costs: In many applications, during a barrier synchronization operation processes are often required to invalidate a large number of pages. Since the cost of an invalidation is high (a single call to `mprotect` is about 10-15 μs), invalidating thousands of pages contributes noticeably to the SVM system time. The primary reason for this relatively high cost is the local TLB flush operation required. To reduce this cost we tried a few schemes and found a simple strategy to help in many of the applications. In our original scheme, we make a single `mprotect` call for every set of contiguous pages to be invalidated. In our new scheme, even when we have a set of non-contiguous pages within a range, if more than a certain fraction of the pages within the range requires invalidations then we invalidate the entire range with a single call. This helps to reduce barrier costs by up to 30% in applications such as LU contiguous and Water spatial. The increase in page faults due to the increased invalidations are quite small in all applications.

Protocol processing: In the implementation, processing can be handled either using a process that occupies a processor and spins, or by having a sleeping process that is woken up by requests from the network. We do not explore the version using a separate polling processor, since we noticed the occupancy of this protocol processor to be quite low.

The cost of protocol overhead can be observed to be low in almost all the application breakdowns presented in Figures 4 and 5.

5 Effect of Clustering

In this section we discuss how the *HLRC* protocol behaves when moving from uniprocessor to SMP nodes. We compare two system configurations for *HLRC*. The first system, *HLRC*, has uniprocessor nodes, whereas the second one, *HLRC-SMP* uses 4-way SMP nodes. In all configurations the speed of the memory and I/O buses is set to 400 MBytes/s and 100 MBytes/s respectively, if we assume a 200MHz 1 CPI processor as discussed earlier. Explanations of the observed performance results are based on detailed examination of simulation data which is beyond the scope of this paper.

When using SMP nodes, the greater the size of the node relative to the total number of processors, the more the communication and synchronization that can be contained cheaply within the node, and the greater the benefits of prefetching, cache-to-cache sharing, and overlapping working sets [7]; at the same time, however, higher demands are placed on the cross-node communication architecture. This is because cross-node communication (per-node, not per processor) does not reduce as quickly as computation time (alternatively, the combined computation power within the SMP node typically increases much faster (linearly) with node size than the degree to which the per processor cross-node communication volume is reduced). We use the simulator to analyze such tradeoffs in great detail.
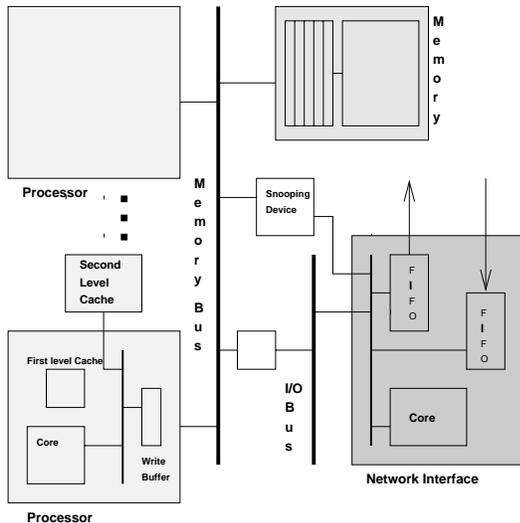
Figure 6: Simulated node architecture.

5.1 Simulated Platforms

For the simulation study of SMP clustering tradeoffs and detailed performance evaluation the simulation environment we use is built on top of augmint [21], an execution driven simulator using the *x86* instruction set and runs on *x86* systems. A more detailed description of the simulator can be found in [2].

The simulated architecture (Figure 6) is similar to the implementation platform. It assumes a cluster of *c*–processor SMPs connected with a commodity interconnect like Myrinet [4]. Contention is modeled at all levels except the network links.

In our simulations, we try to be very close to a realistic system, but we sometimes set parameters to avoid certain artifactual limitations of the real system. For instance the interrupt cost in the simulator is lower than on todays systems, but is implementable. Moreover the simulator does not deal with scheduling artifacts that arise in the real implementation. This approach allows for fewer artifacts in the evaluation, but it does lead to somewhat larger speedups than in the implementation (the breakdowns validate quite well given these differences). The problem sizes that are simulated are of course smaller than real-life problems, and this needs to be taken into account when conclusions are drawn. The simulator provides detailed statistics about all events in hardware, as well as statistics that help identify contention in the various components of the system.

5.2 Simulation Results

Table 4 shows the average number of page faults, page fetches and locks (local and remote) per processor. Similarly, Table 5 presents the average number of messages and bytes sent per processor. Figures 7- 10 present execution costs in the form of bar graphs. The execution time is divided in the following components: *Thread Compute Time*, *CPU Stall Time* (the time the CPU is stalled on local memory references), *Thread Data Wait Time*, *Thread Lock Time* and *Thread Barrier Time*. *I/O Stall Time* gives the time the processor spends on I/O events. The last component,

Application	Problem Size	Speedup	
		HLRC	*HLRC-SMP*
FFT	20	7.73	8.28
LU(contiguous)	512	10.16	12.25
Ocean(contiguous)	514	6.12	12.80
Water(nsquared)	512	8.56	8.84
Water(spatial)	512	7.41	10.05
Radix	1024	0.63	3.41
Volrend	head	7.86	8.81
Raytrace	car	14.06	14.79
Barnes(spatial)	16384	10.94	11.52
Barnes(rebuild)	16384	2.45	3.82

Table 3: Speedups for the uniprocessor and the SMP node configurations.

Application	Messages Sent		MBytes Sent	
	HLRC	*HLRC-SMP*	*HLRC*	*HLRC-SMP*
FFT	7636	2919	10.41	3.95
LU(contiguous)	1014	398	1.61	0.51
Ocean(contiguous)	14183	524	21.25	0.89
Water(nsquared)	5680	2935	1.82	0.51
Water(spatial)	1928	531	3.25	1.01
Radix	10284	3063	64.50	25.11
Volrend	7807	4296	4.39	1.43
Raytrace	9836	3552	11.49	3.42
Barnes(spatial)	2884	1244	12.86	4.11
Barnes(rebuild)	22026	13831	35.47	12.76

Table 5: Number of messages and MBytes sent for each application for the two configurations.

Handler Compute Time, is the time spent in protocol handlers.

From Table 3 we can divide the applications into different classes. The first class is applications for which performance improves noticeably with the use of SMPs. These applications are LU, Ocean-contiguous, Barnes-rebuild, Radix, Volrend and Water-spatial. We differentiate the behavior of these applications in two subgroups.

The first group shows dramatic improvements and consists of Ocean (Figure 7). Table 4 shows that the number of page faults and page fetches is reduced dramatically in the SMP configuration. This is due to the near-neighbor sharing pattern in Ocean. Neighbors are often on the same SMP, so the sharing is much cheaper. Also, the fragmentation (wasted communication) problems observed at column-
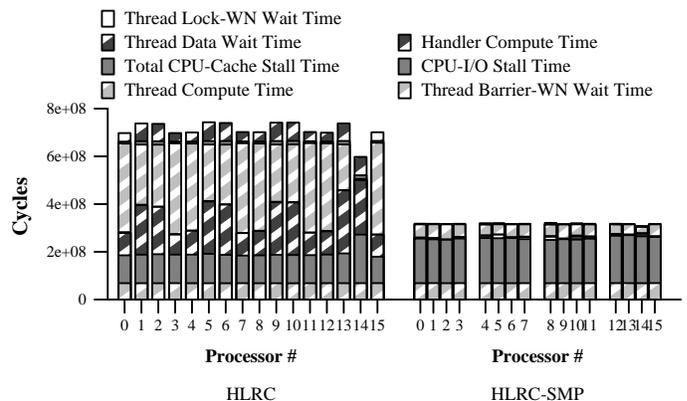

Figure 7: Time breakdown for Ocean (contiguous) for *HLRC*.

Application	Page Faults		Page Fetches		Local Locks		Remote Locks	
	HLRC	*HLRC-SMP*	*HLRC*	*HLRC-SMP*	*HLRC*	*HLRC-SMP*	*HLRC*	*HLRC-SMP*
FFT	2542	1363	2542	972	0	0	0	0
LU(contiguous)	286	150	285	123	0	1	1	0
Ocean(contiguous)	4649	158	4648	155	0	5	15	10
Water(nsquared)	387	132	383	100	0	725	1171	446
Water(spatial)	545	155	543	135	0	7	17	14
Radix	2142	885	2056	473	1	7	47	42
Volrend	1038	489	1033	335	0	192	439	250
Raytrace	2742	783	2742	781	1	68	149	100
Barnes(spatial)	627	158	626	151	0	1	2	1
Barnes(rebuild)	3355	1701	3298	1520	2	529	2229	1698

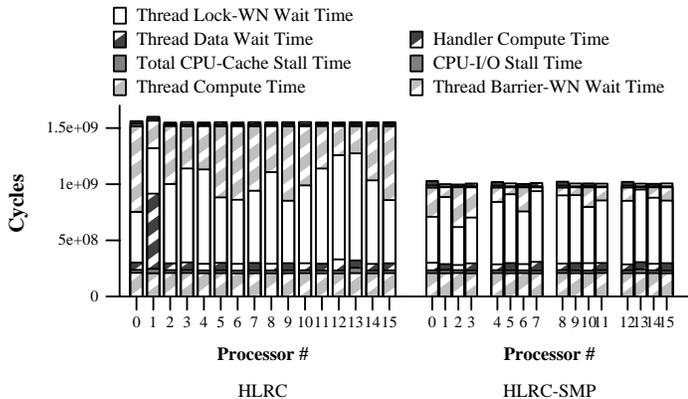Table 4: Average event counts acquired with the simulator for the two configurations.

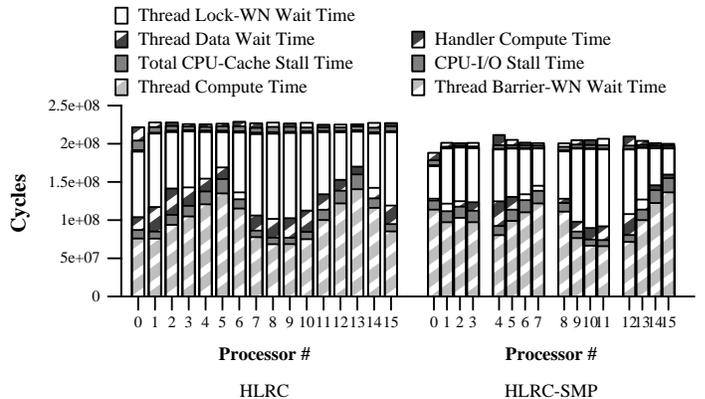Figure 8: Time breakdown for Barnes(rebuild) for *HLRC*.

Figure 9: Time breakdown for Volrend for *HLRC*.

oriented boundaries with uniprocessor nodes [12] are greatly alleviated when the neighbors are in the same SMP. This application uses a large number of barriers to synchronize between phases and time-steps of the computation. The use of hierarchical barriers in *HLRC-SMP* reduces the cost of synchronization dramatically (the simulator shows that it is the barrier cost, not so much load imbalance, that is the problem).

In the second group belong LU, Barnes-rebuild (Figure 8), Radix, Volrend (Figure 9), and Water-spatial (Figure 10), where the improvement comes again from local sharing of data and lower synchronization costs (fewer as well as smaller synchronization and control messages), but at to a much smaller degree than Ocean. Here the inherent communication pattern of the applications does not match the two-level communication architecture as well as in Ocean, but clustering helps nonetheless in data sharing and prefetching. LU benefits mostly from cheaper barrier synchronization. The performance of Radix on the uniprocessor configuration is poor because of a pathological situation in the network interface queues. This disappears in the SMP configuration and performance improves. Volrend is imbalanced because of the different portions of work related to each part of the scene. The version we use does not implement task stealing [13]. Using SMP nodes reduces the imbalances not in computation time but in memory stalls and data wait time. However, although local locks are cheap, remote locks are still expensive in *HLRC-SMP*. The combination of data sharing and cheaper synchronization makes Water-spatial exhibit a big improvement with *HLRC-SMP*.

The second major class of applications consists of FFT, Barnes-spatial, Raytrace, and Water-nsquared. These are applications that do not benefit (or benefit a little) from the use of SMP nodes. In most of these cases, performance is already high and in FFT, other costs are increased due to contention and the effects of sharing, prefetching and cheaper synchronization are overall small.

In general, as can be seen from the execution time breakdowns and the event counts, protocol overheads are reduced by using SMP nodes for the same total processor count. This results in overall gains in performance, especially in the cases where the uniprocessor performance is low.

To further explore the effects of clustering with our protocol we vary the number of processors per node from 1 to 4 to 8, always keeping the total number of processors at 16. An important problem that occurs with clustering is that the useful computation within a node speeds up faster than the communication across nodes is reduced, putting more stress on the now shared note-to-network bandwidth. We conduct two experiments. In the first we keep I/O bandwidth constant at 100MB/s as the number of processors increase, whereas in the second we also increase the I/O and node-to-network bandwidth with cluster size from 100MB/s to 200MB/s to 400MB/s. Figure 11 shows how the speedup varies with cluster size. Each application has three bars that correspond to a configuration with 1,4 and 8 processors per node. The white bars for each application represent speedups as the cluster size increases and the bandwidth remains constant to 100MB/s. The black additions to each bar represent the additional speedup gain as bandwidth increases with cluster size from 100MB/s to 200MB/s to 400MB/s. In cases where applications had poor performance on uniprocessor systems, using SMPs helps substantially. Especially, if bandwidth is increased as well, the per-
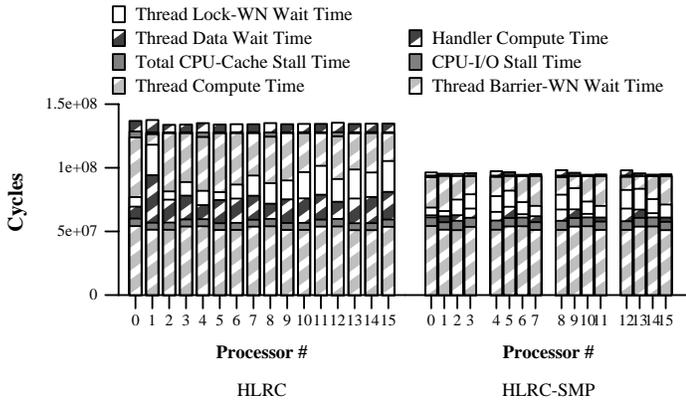
Figure 10: Time breakdown for Water(spatial) for *HLRC*.

formance of applications that are traditionally difficult for SVM (i.e. Radix) becomes decent [2].
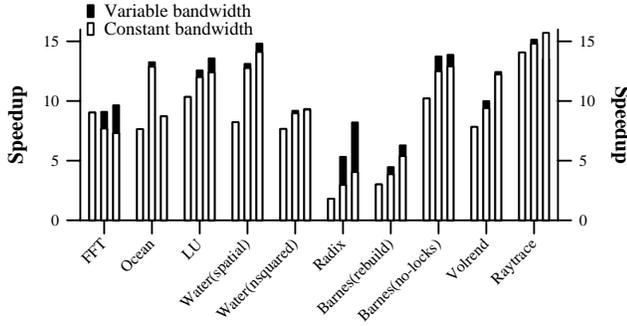

Figure 11: Applications speedups for *HLRC* with increasing cluster size.

6 Related work

Several papers have discussed the design and performance of shared virtual memory for SMPs [5, 14, 8, 18, 22] for different protocols.

HLRC and *LRC* have been compared for some applications on the Intel Paragon in [25]. SVM on SMPs is argued to be a promising direction in [5]. In [14] the authors find that a reserved protocol processor in not required. Earlier versions of this work through simulations can be found in [1, 2].

The SoftFLASH system [8] which has goals similar to us, but uses a single-writer protocol (modeled after the FLASH protocol). The protocol is not as aggressive as other multiple-writer lazy SVM protocols. The system exhibits very high latency and expensive TLB synchronization.

---

[2]For Ocean the best case is always four processors per node. The reason for this is the communication pattern of nearest neighbor and the limited memory bus bandwidth. If the number of processors per node is less than four, then protocol costs are high, because of the communication pattern. If the cluster size is bigger than four, then, although protocol costs do not increase, the contention on the memory bus becomes a problem reducing performance.

The use of 16KB pages in SoftFLASH increases false-sharing and causes fragmentation that result in greater bandwidth needs. We use a multiple-writer home-based protocol based on lazy release consistency (LRC). The network interface we use provides lower latency and higher bandwidth. We use 4KB page and the safe page fetch helps us avoid the TLB synchronization problem.

The Multigrain Shared Memory System (MGS) [24] built on top of the MIT Alewife machine uses a protocol very similar to the SoftFLASH system. The system is implemented by partitioning the Alewife machine, so each node is a distributed rather than centralized shared memory multiprocessor, and the number of external network connections from the node (and hence bandwidth) scales linearly with the number of processors in it. The processors in Alewife are very slow.

The Cashmere-2L system [22] at Rochester is the closest to our system. It uses a multi-writer, directory-based scheme that is an eager variant of release consistency. This is unlike our lazy, vector time-stamp based scheme. The platform is a cluster of Alpha SMPs connected by the Memory Channel network. The protocol takes great advantage of particular features found on the Memory Channel network interface that are not found in other interfaces (such as Myrinet).

The Shasta system [19] performs the coherence in software at cache-line granularity by instrumenting the executable to insert access control operations before shared loads and stores. This idea was first implemented in the Blizzard-S system [20]. There is currently an implementation of Shasta that runs on a cluster of Alpha SMP nodes [18]. This implementation does show promising performance. However, it is not page-based shared virtual memory. It has control-data structure requirements that are dependent on the number of processors which may lead to scalability problems for large number of processors. Also, it relies on code instrumentation to detect accesses to share data.

7 Discussion and Conclusions

We have presented the design and implementation of a lazy, home-based, multiple-writer SVM protocol across SMP nodes that exploits the hardware coherence and synchronization within an SMP, yet provides lazy consistency even within an SMP, and discussed some important tradeoffs and our choices. We have also presented the key data structures that simplify implementing different choices, and the partitioning of data structures between per-process and per-node.

The results from the actual implementation of the protocols are very promising and demonstrate good performance. The execution breakdowns exposes the bottlenecks in applications that do not perform well. These help in pointing us in the direction for further improvements in performance, though some of these may require major changes in our protocol design. Also, some bottlenecks seen may be due to certain characteristics inherent in the application.

In comparing 4-way SMP nodes versus uniprocessor nodes for the same total processor count (through detailed architectural simulations), we find that performance improves substantially with the use of SMPs in six out of ten very different applications. The other four show a smaller improvement. All applications improve further when 8-way SMP nodes are used, except Ocean where the prob-

lem becomes saturation of the memory bus inside each node. Among the irregular applications we examine, Barnes-rebuild, Radix, Volrend and Water-spatial benefit substantially and Barnes-nolocks, Raytrace and Water-nsquared do not exhibit any significant improvement. Overall, we found we found detailed simulations to be an invaluable tool in understanding system behavior and application-system interactions that are difficult to obtain from the real implementation, and in comparing with the real implementation results to diagnose the causes of conflicts in the latter.

The results on the real implementation are quite similar to those as the simulator, though a little worse due to the differences in interrupt cost, OS scheduling problems and some other factors. We would like to extend our system to larger scale systems and larger problem sizes to see how well *HLRC-SMP* can be used to extend the shared address space programming model across SMP clusters.

8 Acknowledgments

References

[1] A. Bilas, L. Iftode, D. Martin, and J. Singh. Shared virtual memory across SMP nodes using automatic update: Protocols and performance. Technical Report TR-517-96, Princeton, NJ, Mar. 1996.

[2] A. Bilas, L. Iftode, and J. P. Singh. Comparison of shared virtual memory across uniprocessor and SMP nodes. In *IMA Workshop on Parallel Algorithms and Parallel Systems*, Nov. 1996.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[5] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 106–117, Apr. 1994.

[6] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, April 1997.

[7] A. Erlichson, B. Nayfeh, J. Singh, and K. Olukotun. The benefits of clustering in shared address space multiprocessors: An applications-driven investigation. In *Supercomputing '95*, pages 176–186, 1995.

[8] 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*, pages 210–220, October 1996.

[9] R. Gillett. Memory channel network for pci. In *Proceedings of Hot Interconnects '95 Symposium*, Aug. 1995.

[10] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[11] L. Iftode, J. 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.

[12] L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[13] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Sixth ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[14] M. Karlsson and P. Stenstrom. Performance evaluation of cluster-based multiprocessor built from atm switches and bus-based multiprocessor servers. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[15] P. Keleher, A. 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, Jan. 1994.

[16] P. Keleher, A. 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.

[17] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-based shared memory on low-latency, remote-memory-access networks. *Proc., 24th Annual Int'l. Symp. on Computer Architecture*, June 1997.

[18] D. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Oct. 1997.

[19] D. Scales, K. Gharachorloo, and C. 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*, Oct. 1996.

[20] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. 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, Oct. 1994.

[21] A. Sharma, A. T. Nguyen, J. Torellas, M. Michael, and J. Carbajal. Augmint: a multiprocessor simulation environment for intel x86 architectures. Technical report, University of Illinois at Urbana-Champaign, March 1996.

[22] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, Oct. 1997.

[23] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1995.

[24] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: a multi-grain shared memory system. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[25] 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*, Oct. 1996.