Improving Release-Consistent Shared Virtual Memory using Automatic Update

Liviu Iftode, Cezary Dubnicki, Edward W. Felten and Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

Shared virtual memory is a software technique to provide shared memory on a network of computers without special hardware support. Although several relaxed consistency models and implementations are quite effective, there is still a considerable performance gap between the "software-only" approach and the hardware approach that uses directory-based caches.

Automatic update is a simple communication mechanism, implemented in the SHRIMP multicomputer, that forwards local writes to remote memory transparently. In this paper we propose a new lazy release consistency based protocol, called Automatic Update Release Consistency (AURC), that uses automatic update to propagate and merge shared memory modifications. We compare the performance of this protocol against a software-only LRC implementation on several Splash-2 applications and show that the AURC approach can substantially improve the performance of LRC. For 16 processors, the average speedup has increased from 5.9 under LRC, to 8.3 under AURC.

1 Introduction

Since the idea of shared virtual memory (also called distributed shared memory) was proposed, it has attracted great interest in the systems and architecture communities, because it is an inexpensive alternative to hardware shared-memory approaches. Shared virtual memory offers cost advantages over hardware-only approaches because it can run on a network of computers without physically shared memory, but the cost savings comes at some cost in performance, especially for applications with fine-grained sharing. The challenge is to find a method that brings the performance of the full-hardware approach at the low cost of shared virtual memory.

Our approach to this problem is to consider adding some simple, inexpensive hardware to a workstation network, and then redesigning a shared virtual memory algorithm to exploit this hardware. Since the software will handle many of the details of the coherency protocol, the hardware can be relatively simple; since the hardware will handle the common cases, the mechanism as a whole can be relatively fast.

Several recent studies indicate that a key to closing the performance gap is cooperation between the network interface and shared virtual memory software systems. Recent studies show that the overhead of coherence protocol messages can affect the performance of a "software-only" shared memory system significantly [14, 7], and that on distributed memory architectures with remote memory reference capability, the performance of software cache coherence maintained at the virtual memory page level is competitive with that of hardware cache coherence schemes [23, 24, 15]. These results suggest that appropriate architectural support may improve the performance of shared virtual memory substantially.

The SHRIMP project at Princeton studies how to provide high-performance communication mechanisms to integrate unmodified, commodity desktops such as PCs and workstations into inexpensive, high-performance multicomputers. One of our research goals is to investigate ways to improve the performance of the "software-only" shared virtual memory approach by using appropriate consistency models and by providing necessary, yet simple, support at the network interface level.

Our main idea is to take advantage of the automatic update feature of a virtual memory mapped network interface to improve the performance of shared virtual memory. *Automatic update* is a very simple, point-to-point communication mechanism that propagates updates to mapped destinations directly without the intervention of software. We take advantage of this simple mechanism in two ways: supporting fine-grained updates to shared data and reducing coherence protocol overheads. To accomplish this task, we have developed a shared virtual memory coherence protocol based on the lazy release consistency model.

To understand the performance implications of this approach, we have implemented both our method and the "software-only" shared virtual memory approach within the TangoLite simulation framework [11]. We conducted our simulation studies with the same architectural and systems configurations except that our approach uses the automatic update feature of the network interface. The performance results with several Splash-2 benchmark applications show that on a net-

In Proceedings of the 2nd International Symposium on High-Performance Comp uter Architecture, February 1996.

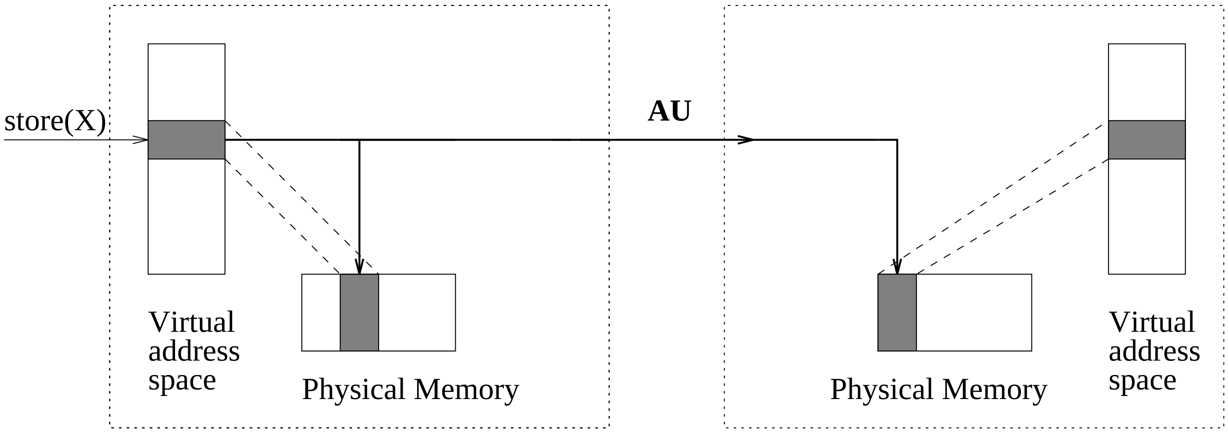

Figure 1: The Automatic Update Scheme.

work of 16 processors, our method (called Automatic Update Release Consistency) outperforms lazy release consistency by factors of 1.15 - 2.5.

2 Automatic Update

A key to reduce the overhead of consistency protocols is to reduce the overhead of message-passing primitives. The SHRIMP multicomputer, currently being built at Princeton, aims to reduce message-passing overhead by designing new network interface hardware. Our recent research shows that the "virtual memory mapped communication" technique used in SHRIMP can reduce the overhead of sending a message to a few user-level instructions [4, 3]. The computing nodes are Pentium PCs running the Linux operating system. This architecture allows selecting write-through or write-back caching mode per page basis. The interconnection is a comercially available network which delivers packets reliably in FIFO order. The SHRIMP team has built a network interface card to connect the PCs to the routing backplane, and the software necessary to use the system effectively. SHRIMP supports two modes of transferring data: deliberate update and automatic update.

Deliberate update is an explicit transfer mechanism that allows a process to copy a region of its memory into the memory of a remote process. Deliberate update can be thought of as a lower-overhead substitute for traditional message-passing.

This paper will focus instead on the implications of the *automatic update* mechanism (Figure 1). In automatic update, a *mapping* can be created from a virtual page of a "source" process to a virtual page of a "destination" process. Once a mapping exists, all writes to the source page will automatically be propagated, or "written through," to the destination page.

To ensure security, mappings must be set up by the operating system; but once a mapping exists, writes are propagated automatically by the hardware. Any page of local, cacheable memory can be the source or destination page of a mapping. While each page of local memory can be mapped to a different remote page, writes to the same local page must be propagated to a single destination page. Mappings go in one direction — there is no transfer from destination to source — but a separate mapping in the reverse direction can be set up if desired.

Automatic update is implemented by having the network interface snoop all write traffic on the memory bus. Pages with outgoing automatic update mappings use write-through caching, so that every CPU write to such a page causes a transaction on the memory bus. On seeing a write, the network interface hardware checks to see whether the page being written has an automatic update mapping; if it does, the network interface automatically creates a network packet and sends it to the correct destination. The network interface also combines automatic updates to consecutive addresses into a single packet to reduce network traffic. The automatic update packets know their destination physical memory addresses, so the updates can be transferred directly to destination memory without interrupting the CPU.

Automatic update has two important features: low initiation overhead and overlapping communication with computation. Since automatic update is implemented by snooping memory writes off the memory bus, the cost of initiating an update is a regular memory store instruction to local memory. In the absence of CPU write buffer stalls, there is no processor overhead, since the transfer is initiated by a store instruction which the program would execute anyway. The network interface generates and sends out a network packet while the CPU executes subsequent instructions.

3 Lazy Release Consistency

Lazy release consistency [14] (LRC) is a variant of release consistency [9] (RC). Release consistency guarantees memory consistency only after explicit synchronization points, allowing the SVM system to hide the latency of update propagation by using either buffering or pipelining techniques. The consistency protocol requires programs to be properly labelled by explicitly marking all synchronization points either as acquire or release operations. For any synchronization variable s, one processor's *release(s)* followed by another processor's *acquire(s)* causes the system to guarantee that all updates to shared memory made by the releasing node become visible to the acquiring node (Figure 2).

In lazy release consistency, the propagation of modifications to shared memory is postponed as long as possible—until an acquire is performed. At this time, the acquiring processor determines which modifications it needs to see according to the definition of release consistency.

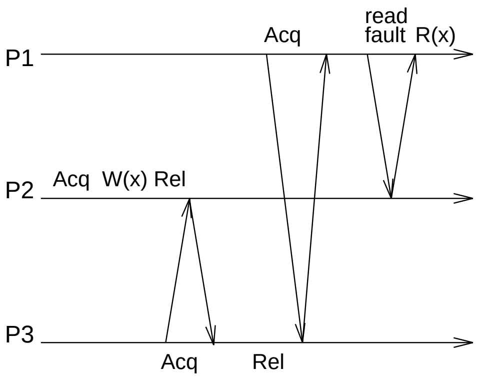

Figure 2: Actions of LRC Protocol. Each arrow represents a message. *P2* modifies x and releases a lock, which is passed to node *P3* and then to *P1*, which reads x. *P1*'s read faults, causing a diff to be fetched from *P2*.

The LRC protocol divides the time on each node into intervals. Each node maintains a scalar local clock which always holds its current interval number. Every time a node executes a release or an acquire, a new interval begins and the clock is incremented. Acquire and release operations establish causal order [16] among intervals on different nodes. This order is kept by assigning to each interval a *vector timestamp*, with one entry for each node.

The LRC protocol associates updates to shared pages with vector timestamps to minimize data transfers while ensuring the correctness of release consistency. When a processor writes a page, it creates a *write notice* for that page; the local timestamp of the interval when the page was written is put into the write notice. Thus, the write notice records the fact that the page was written during that specific interval.

Every processor maintains a list of all the write notices of which it is aware. When a processor acquires a lock, it sends the previous lock holder a copy of its current vector timestamp. The previous lock holder responds by sending the write notices corresponding to the intervals missing at the receiving node. The processor acquiring the lock receives these write notices and remembers them; it sets its vector timestamp to the elementwise maximum of its previous timestamp and the timestamps of all of the received write notices. It also invalidates all of the pages for which it receives write notices, because the arrival of a write notice means that the page has been written since the lock-acquiring processor last got an up-to-date copy of the page.

To get up-to-date copies of data, the LRC protocol computes the updates in software. When a processor first writes a page, it makes and saves a copy of the page. At the time a lock is released, the processor compares the current copy with the saved copy. The result of the comparison is a list of updated locations and their new values, called a *diff*. The processor associates the diff with a write notice.

When a processor needs to get an up-to-date version of a page, it consults its list of write notices to determine which diffs it needs in order to bring its copy of the page up to date. It then fetches the diffs (Figure 2) from other nodes and applies the diffs to its copy of the page in the proper causal order.

Further details about lazy release consistency implementations can be found in [8] and [13].

4 Automatic Update Release Consistency

In this section we describe three protocols which use automatic update to implement lazy release consistency. The first protocol, *Copyset-2*, allows only two nodes to share a page at any given time, i.e. the copyset size of a page is less than or equal to two. The second protocol, *Copyset-N*, allows arbitrary copyset sizes. The third protocol, Automatic Update Release Consistency (*AURC*), is a hybrid of the first two protocols. The first two protocols are described for illustration only, to allow better understanding of *AURC*, which is the protocol we advocate for SVM systems with automatic update.

The key performance issue for any software implementation of shared memory is to support fine-grained sharing with minimum cost. To achieve this goal, a software shared memory system must efficiently:

1. detect writes to shared locations,
2. prevent access to outdated versions of data, and
3. get up-to-date copies of data when necessary.

The common approach to the first problem is to use the virtual memory translation hardware to field access page faults in order to maintain memory coherence in system software. All of our protocols use this standard method to detect first writes to shared data.

In solving the second problem, our protocols are similar to LRC: both protocols use scalar logical clocks and vector timestamps to prevent accesses to outdated versions of data. The difference is that our protocols do not need vector timestamps to preserve the partial ordering among the write-notices.

The novel feature of these protocols is how they get up-to-date copies of data when necessary. Instead of computing diffs as LRC does, our protocols take advantage of automatic update to propagate and merge shared memory modifications. In our ptotocols the updates can be propagated in two ways:

1. Use the automatic update mappings to forward local writes to a remote destination

2. Fetch on demand to update a local copy of the page at the time of page fault.

In order to ensure that updates are completed before pages are either accessed or fetched on demand, it is sufficient to flush the data links through which automatic updates are propagated. This is true because we assume that the network delivers the messages in the order that they are sent.

For this purpose we introduced two sets of vector timestamps, called *flush timestamp* and *lock timestamp.* At each node, a flush timestamp vector is attached to each in-memory page. Its role is to record which updates the processor has seen so far for that particular page. A lock timestamp vector is kept for each shared page at each node. It records which updates the processors is required to have in place for a given page, before it can access that page. The information the lock timestamps provide is the same as that the write-notice lists provide in LRC. To speed up the lock acquire each processor keeps also a global lock timestamp which is elementwise maximum of all per-page lock timestamps of that processor.

If an element in the flush timestamp vector for a page is smaller than the corresponding element in the page's lock timestamp vector, it means that the processor's version of the page is out of date. If an element of the flush vector is larger than the corresponding element in the lock timestamp, it means that the processor has a more recent copy of the page than the one needed according to release consistency rules.

4.1 Copyset-2 Protocol

The Copyset-2 protocol maintains the invariant that no more than two processors have a valid copy of a shared page at any time. For a page shared by two nodes, we create mappings for automatic update between the two pages bidirectionally as shown in Figure 3. Updates to the shared page from one node will be propagated to the other node automatically. When updates complete, both nodes have the most up-to-date copy.

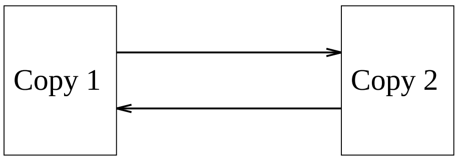

Figure 3: Automatic Update Mappings for a Page Shared by Two Nodes.

If a third node accesses a page already shared by two nodes, the copyset is adjusted by invalidating one of the two existing copies. The protocol then makes a new page copy on the requesting node, and creates a bidirectional mapping between the new copy and the remaining valid copy.

To implement the lazy release consistency model, one needs to guarantee the ordering of events when nodes are performing acquire and release operations. The main complication is that an indirect chain of causality may exist. Assume that node $P1$ and $P2$ share a page and that there is a chained lock acquisition from $P2$ to some node $P3$ and then to $P1$. Although $P2$ has never passed a lock directly to $P1$, $P1$ needs to see $P2$'s updates to the shared page. Our protocol must ensure that updates are properly propagated in the presense of arbitrary network delays.

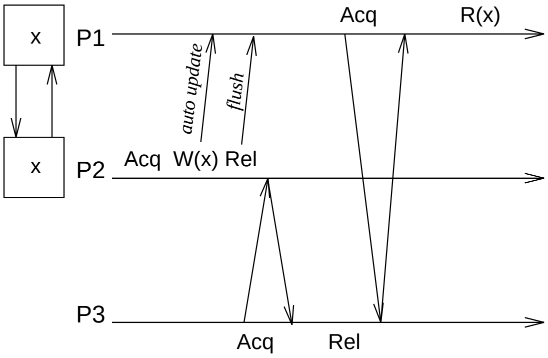

Figure 4: Actions of Copyset-2 Protocol. Each arrow represents a message. Page x is shared by nodes *P1* and *P2*. *P2* modifies x, causing an automatic update, then releases a lock, causing a flush message. The lock is passed to node *P3* and then to *P1*, which reads x.

On a release, a processor increments its logical clock and uses it to update via automatic update the corresponding entry in flush timestamp vectors of the second copies which have been written during the last interval. In this way, all data links which were active during the last interval are flushed.

Similarly to LRC, when a processor acquires a lock, it sends the previous lock holder a copy of its current global lock timestamp vector. The previous lock holder responds with the set of write-notices for all pages which are out-of-date on the acquiring node. The acquiring processor receives this message, and sets the lock timestamp of each of its pages to the elementwise maximum of the page's previous timestamp, and the timestamp for that page received in the write-notice.

If every component of a page's new lock timestamp is less than or equal to the corresponding component of its flush timestamp, then the processor knows it still has an up-to-date version of the page. If not, the local copy of the page is out of date, so the acquiring node unmaps the page, but keeps its contents in place. Automatic updates destined for the page are in progress, and they will arrive eventually, bringing the page up to date.

On a page access miss, if the faulting processor already has a copy of the page, it stalls until every element of the page's flush timestamp is at least as big as the corresponding element of its lock timestamp. In most cases, page unmapping will not be necessary, as all of the flush messages will have arrived before the lock acquisition. Therefore, this type of page fault will be rare in Copyset-2.

Figure Figure 4 shows an example of the actions of Copyset-2 protocol. The main advantage of the Copyset-2 protocol is that there is no need to compute diffs, since updates are propagated automatically. The main disadvantage is the potential increase in the number of page invalidations due to the limited copyset size.

4.2 Copyset-N Protocol

The Copyset-N protocol, as its name suggests, supports an arbitrary number of copies of a page. The protocol uses a designated node as the *owner* of a shared page, and establishes automatic update mappings from other nodes sharing the page to the owner.

Figure 5 shows the mappings from copy holders to an *owner* for a shared page. When copy holder nodes write to their own local copies of the shared page the updates are propagated automatically to the owner as well. As a result, the automatic update mechanism merges updates from multiple writers without computing diffs.

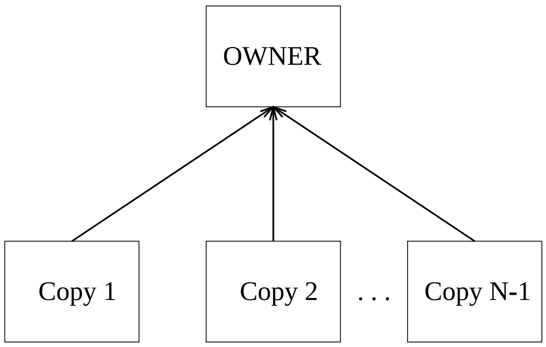

Figure 5: Automatic Update Mappings for a Page Shared by More Than Two Nodes.

Similar to the Copyset-2 case, the Copyset-N protocol also needs to flush automatic update links before performing a release operation. Thus, on a release, a processor performs the same steps as in the Copyset-2 protocol, except that flushes with the logical clock are sent to the owners of the updated pages (Figure 6).

On an acquire, the actions of the Copyset-N protocol are similar to those of Copyset-2: out-of-date pages are invalidated.

The important difference between the two protocols is in using the fetch mode. In Copyset-2, a faulting access after a page invalidation is solved by waiting until all updates to the page have arrived. In the Copyset-N protocol a faulting access occurring at a non-owner processor processor requires to *always* fetch a copy of the page from the owner, in order to get the necessary updates (though it is possible to prefetch). At the same time, the faulting processor also receive from the owner the full flush timestamp vector for that page.

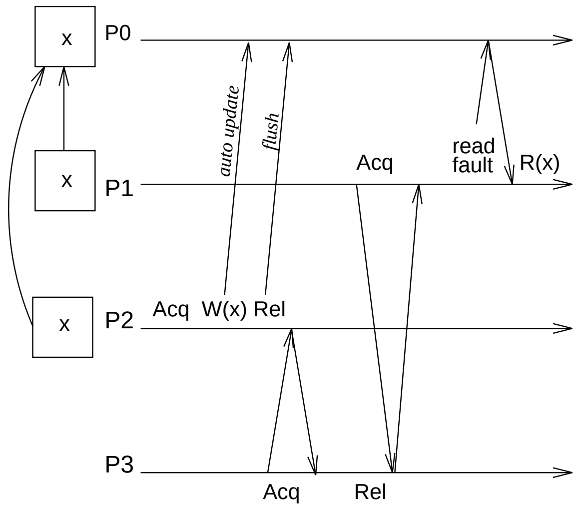

Figure 6: Actions of Copyset-N Protocol. Each arrow represents a message. Page x is shared by nodes *P0*, *P1* and *P2*; *P0* is the owner of x. Node *P2* modifies x, causing an automatic update, then releases a lock, causing a flush message. The lock is passed to some node *P3* and then to *P1*, which reads x. *P1*'s read faults, causing an up-to-date version of the page to be brought over from the owner.

The main advantage of the protocol is that it may have fewer page invalidations compared to Copyset-2. The drawback is that Copyset-N does not take full advantage of the automatic update mechanism, which motivates the following hybrid protocol.

Compared to LRC, Copyset-N takes fewer coherency-related page faults, because no faults are required at the owner node. The possible drawback is that Copyset-N relies on hardware to updates which might lead to an increase in message traffic.

4.3 AURC: A Hybrid Protocol

The AURC protocol combines Copyset-2 and Copyset-N in a straightforward way. The main idea is to use the Copyset-2 protocol for pages shared by two nodes, and to fall back on the Copyset-N protocols for all other pages.

Initially, the Copyset-2 protocol applies to all pages shared by two nodes. If a third node accesses the page, the page will be thereafter managed by the Copyset-N protocol. We call this hybrid protocol AURC because it takes full advantage of the automatic update mechanism.

The hybrid protocol has the advantages of both Copyset-2 and Copyset-N. The key advantage over the software-only LRC approach using the traditional network interface is that it eliminates the need to make mirror copies and compute diffs. The automatic update mechanism supports fine-grained updates to shared data. In addition, while applying the Copyset-2 part of the protocol, eager release consistency can be maintained without page transfers.

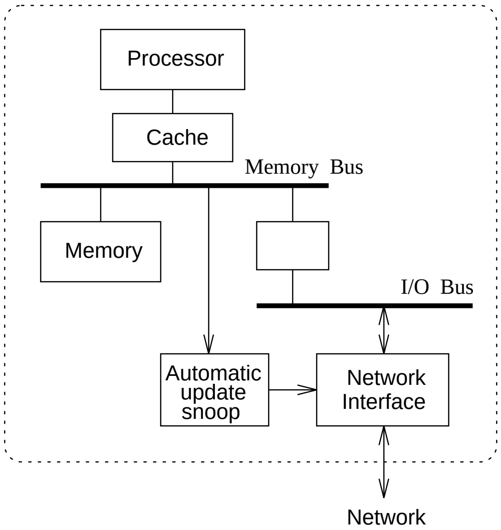

Figure 7: Architecture of SHRIMP Node.

5 Performance

To evaluate the performance of our SVM protocols we implement them on a SHRIMP architecture simulator built within the TangoLite execution-driven multiprocessor simulation framework [11].

5.1 The Architectural Model

We consider a basic architectural model close to the SHRIMP multicomputer architecture with commodity PC Pentium boards as nodes (Figure 7).

Parameter	Value
Processor clock	60 MHz
Page size	4 Kbytes
Data cache	256 Kbytes
Cache line size	32 bytes
Write buffer size	4 words
Memory bus bandwidth	80 Mbytes/sec
Page transfer bandwidth	28 Mbytes/sec
Trap cost	150 cycles
Message send latency	5 μsec
Page copying	4 cycles/word plus cache misses
Diff calculation	4 cycles/word plus cache misses
Diff application	8 cycles/word plus cache misses

Table 1: System Parameters Used in Simulation.

Table 1 gives the overview of the architectural parameters used in our simulations. We assume the same clock rate (60 MHz) as in SHRIMP, and enough main memory to fully accommodate the entire application address space. The data cache is write-around, two-way set-associative and features a per-page selectable write-back or write-through policy.

Local memory bus arbitration between the processor and the SHRIMP network interface occurs every 64 bus cycles. The processor always has priority over the SHRIMP interface but once the processor's request is posted it may still have to wait up to 64 bus cycles before the bus is granted. The SHRIMP interface also has an internal arbitration between incoming and outgoing data transfer, with incoming given higher priority.

5.2 Protocols

To evaluate the performance of the proposed SVM system using automatic update support we implemented the LRC, Copyset-N, and AURC described in section 3. To ensure a fair comparison in evaluating the performance impact of automatic update for SVM, we use the same processor and network architecture for all three protocols. The only difference is that AURC can use both automatic update and deliberate transfer modes, whereas LRC can use only deliberate transfer.

AURC maps as write-back all non-shared pages and write-through all shared pages. In LRC, all pages are mapped write-back since only deliberate update is used.

We used the SHRIMP network interface message-combining feature to combine consecutive messages both at the sender and the receiver. At the sender, automatic updates to consecutive addresses within the same page are coalesced together into a single message. At the receiver, consecutive messages which fit in the same arbitration interval are delivered to memory in a single bus transaction.

Global memory allocation is always aligned on a page boundary. but the SVM system processing and transfer is limited to only the allocated size.

For both protocols, a fast trap mechanism [26] to handle notification on asynchronous message arrival is assumed.

5.3 Applications

To evaluate the performance of our SVM protocols, we chose to run a subset of Splash-2 [28] applications and kernels, covering both regular(LU, Water and Ocean) and irregular (Cholesky) sharing.

LU is a kernel which factors a 512 x 512 dense matrix into the product of a lower triangular and an upper triangular matrix. To enhance data locality and to avoid false sharing, the matrix is factored as an array of 32 x 32 element blocks which are statically assigned to processors. Barriers are used to ensure synchronization between the processor producing the pivot element and those consuming it.

Water-nsquared is a molecular dynamics simulation, similar to the original Water code from the Splash-2 suite. It computes intermolecular interactions for a system of 512 molecules for 5 steps. Each processor is statically assigned a number of molecules. In each iteration, a processor computes the interaction between every molecule in its partition and n/2 other molecules. Updates are accumulated locally between

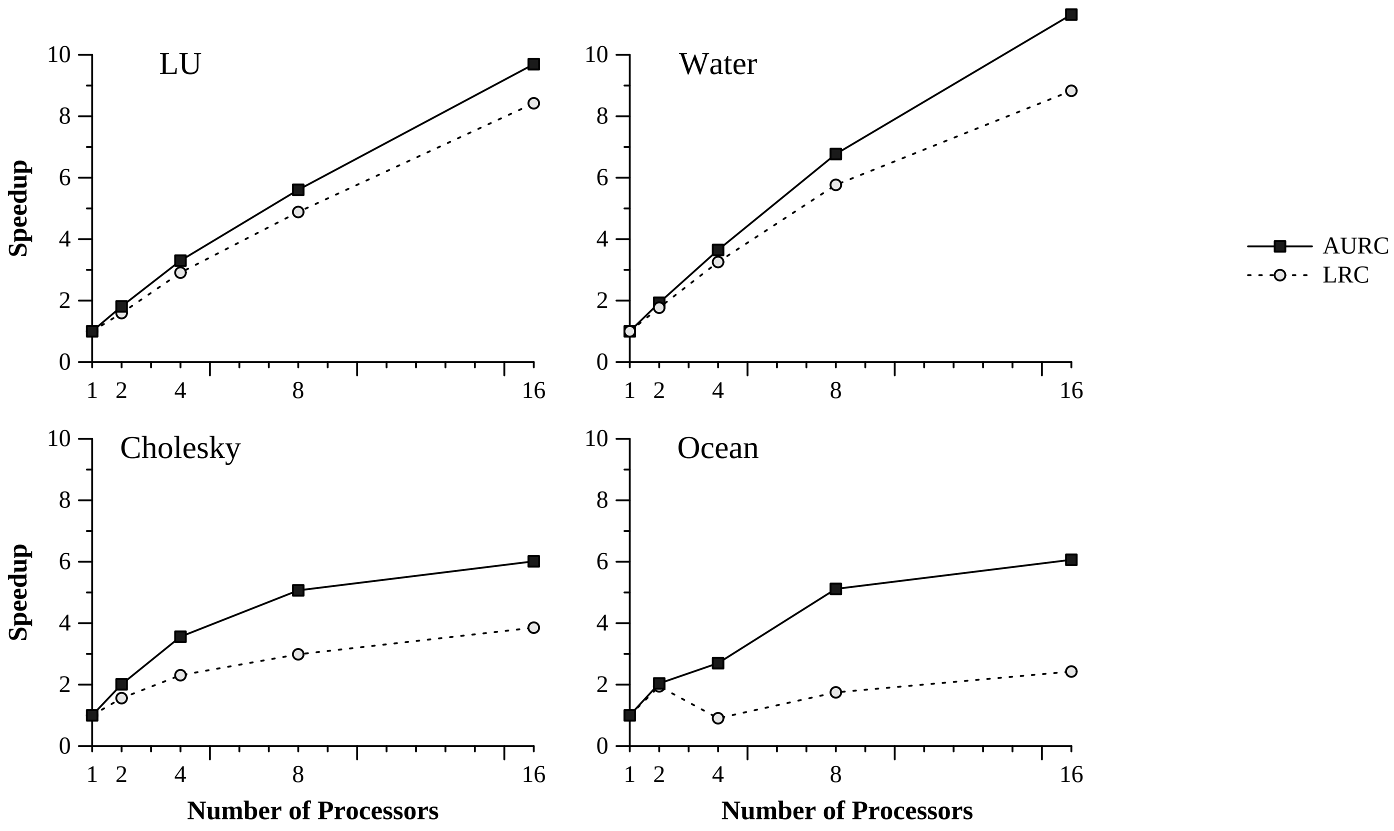

Figure 8: Speedups for LRC and AURC protocols.

iterations, and performed at once at the end of each iteration. Distinct locks are used to protect data for each molecule.

Cholesky is a kernel which performs blocked Cholesky factorization of a sparse positive definite matrix. It is a program with fine-grained sharing and a high communication to computation ratio. This explains its relatively poor performance even on hardware shared-memory machines. Synchronization is handled mostly with locks. We ran Cholesky with the tk29 input file.

Ocean is a fluid dynamics application which simulates large scale ocean movements. Spatial partial differential equations are solved at each time-step using a restricted Red-Black Gauss-Seidel Multigrid solver. We simulate a square grid of size 514 by 514 points with the error tolerance for iterative relaxation set to 0.001.

Ocean is a representative of the class of problems based on regular-grid computation. At each iteration the computation performed on each element of the grid requires the values of its four neighbors. Work is assigned statically by splitting the grid among processors. Nearest neighbor communication occurs between processors holding adjacent blocks of the grid. We chose to simulate an Ocean implementation with contiguous partition allocation using four-dimensional arrays for grid data storage to avoid false sharing [28].

5.4 Speedups

Figure 8 shows the speedups for the AURC and LRC protocols, for all four programs. In all cases, AURC outperforms LRC: by a factor of 1.15 for LU, 1.28 for Water, 1.57 for Cholesky and 2.5 for Ocean.

The performance improvement of AURC over LRC increases with the amount of sharing a program exhibits, expressed as a ratio of communication to computation.

Cholesky and Ocean require significant network bandwidth, with an order of magnitude larger communication to computation ratio than LU and Water (Figure 9). In both cases, the Copyset-2 protocol is the key which allows AURC to perform much better than LRC. The price paid by AURC (increased processor stalling, and increased data transferred, as seen in Figure 10) is substantially smaller than the price LRC has to pay to produce and apply diffs.

On the other hand, LU and Water have relatively small communication to computation ratio (Figure 9). As a result, any improvements introduced by the AURC protocol have a smaller impact on overall performance (Figure 10).

5.5 Comparative Analysis of AURC and LRC Performance

In this section we discuss in detail factors contributing to the performance of LRC and AURC.

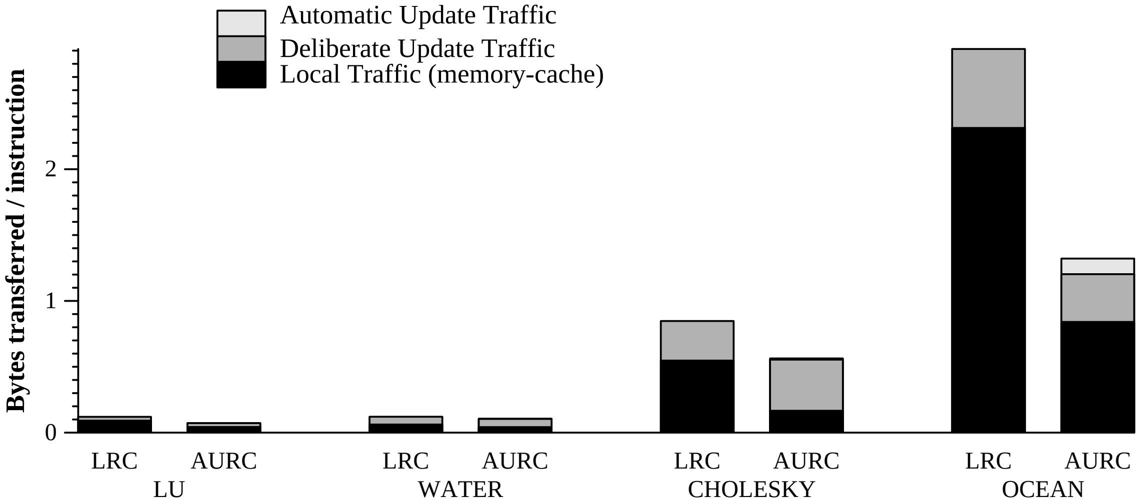

Figure 9: Memory Bus Data Traffic (16 processor case).

5.5.1 Cost of Sharing

Cost of sharing provides a good metric to compare the intrinsic benefits of the two protocols while factoring out the computation and synchronization cost. The cost of sharing can be roughly estimated as the cost per page fault times the number of page faults. In table 2 we report the cost of sharing per page fault and the number of page faults for the set of applications we studied. In almost all cases, the cost per page fault is about 3 times higher in LRC than in AURC.

Application	Cost per Fault (CPU cycles * 1000)		Number of Page Faults	
	LRC	AURC	LRC	AURC
LU	61	18	99	99
Water	57	18	467	395
Cholesky	38	23	4770	3841
Ocean	96	34	4553	3364

Table 2: Number of Page Faults per Processor and Cost of Sharing per Page Fault (16 processors).

AURC reduces both components of the cost of sharing. The cost of a page fault can be reduced because AURC avoids all diff-related operations while employing automatic update to provide the same functionality. The number of page faults is also reduced, because when a page is managed by the Copyset-2 protocol, no page faults occur. Also, for a page managed by Copyset-N, no page faults occur at the owner processor.

The cost of sharing per page fault in AURC is the cost of automatic updates plus the cost of page transfer from the owner of the page to the faulting node.

In the LRC case, the cost of sharing has two components: a processing cost and a data transfer cost. The processing component includes the cost of producing a diff, which is amortized over the number of processors which use that diff, plus the cost of applying the diff at the consumer sites. The total cost of sharing per page fault includes aggregation of both components for all diffs necessary to bring the faulting page up to date.

Under the given architectural model, the minimum cost of producing a diff (about 8000 processor cycles — see Table 1) is only slightly less than the cost of page transfer over the network (about 9000 processor cycles). This is caused by low effective message bandwidth relative to processor speed.

Although the cost of producing a diff can be amortized over many page faults, in many cases (Water, Ocean), the ratio of diffs produced to number of page faults can be close to one (or even greater than one in some cases, due to false sharing). Consequently, the AURC transfer cost is almost covered by the LRC cost of calculating diffs. In addition, LRC has to pay the cost of memory accesses caused by diffs as well as for the diff transfer itself.

LU exhibits one-producer multiple consumer sharing. In such a case, the cost of producing diffs can be amortized over several page faults after each release. Nevertheless, there is still a processing cost in applying the diff at the faulting site which increases the cost of each page fault in LRC.

When a diff is transferred by LRC, not only the

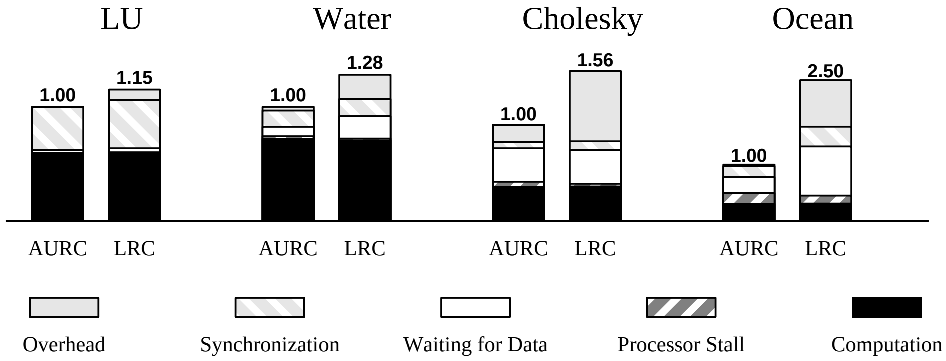

Figure 10: Execution Time Breakdown for 16 Processors. Execution time is normalized to AURC.

modified words are sent, but also their page offsets. As a result, for medium and coarse grained sharing the transfer cost per diff approaches the cost of a full page transfer. Performance of LRC is further reduced if multiple diffs must be transferred to update the faulting page, even when all of the diffs overlap or are found at one site. Additionally, false sharing can also create multiple "producers" on the same page. By sending diffs instead of full pages, LRC can win over AURC in terms of transfer cost per page fault, but only for small updates and only when a page fault does not require many diffs.

In terms of the number of page faults, AURC wins over LRC in most cases (Table 2). One explanation for this is that AURC avoids the page faults at the owners since the updates are automatically propagated to them. But, the main saving in the number of page faults is a direct consequence of including Copyset-2 in AURC. Pages shared by two nodes are kept updated on both sites through automatic updates without any page faults. Since the transition from Copyset-2 to Copyset-N is done dynamically, the later it happens, the more page faults AURC can save compared to LRC. Sharing by two nodes could be either a stable property of the program (true sharing as in contiguous Ocean), or it could be an artifact of a given input data and number of processors (as in Cholesky). In the former case the performance improvement is always significant, while in the latter it may vary significantly with the input parameters.

From the above analysis we can deduce that if bus contention can be ignored then sharing is more expensive in LRC than in AURC.

5.5.2 Bus contention

Although automatic updates increase network bandwidth requirements in AURC (Figure 9), the overall cost of sharing is still much smaller than for LRC for the applications we tested (see cost per fault, Table 2).

Automatic updates do not come for free, and for an application like Ocean, network bandwidth and local memory bus cycles are heavily used to update copies of pages. Thus we observe a significant processor stall time for AURC (see Ocean in Figure 10). But this cost is acceptable because of the savings in the total number of page faults (see Ocean in Table 2).

In fact, the amount of data transferred with automatic update can be reduced by either programmer or SVM system solutions. Although a careful programmer can reduce the bandwidth requirement by using local variables to store intermediate results, in our AURC simulations we did not take advantage of this opportunity; we relied on solutions involving only the SVM system.

On the other hand, LRC diff processing also causes large local memory traffic due to additional cache misses (3-4 times more for Cholesky and Ocean in Figure 9). This slows down the traffic to and from the network.

In summary, although AURC causes more network traffic than LRC (up to 30% more for the applications we studied, see Figure 9), AURC generates less contention on the local memory bus than LRC. As a result, the cost of sharing is smaller in AURC, even if we consider bus contention.

5.5.3 Cost of Synchronization

For the applications we ran, the cost of synchronization is similar in both protocols and does not significantly affect the differences in their performance. The only exception is Ocean, when the difference in the number of page faults increases barrier cost for LRC (Figure 10).

Both point-to-point (lock) and global (barrier) synchronization primitives are used in the applications. Point-to-point synchronization can be implemented more effectively than barriers, in both the AURC and LRC protocols. However, global synchronization is extensively used in all applications except Water and Cholesky, either when necessary to mark a one-to-many data release as in LU, or, more often, just as a programming convenience (Ocean).

Synchronization, in particular global synchronization, introduces a serious threat to SVM performance particularly if implemented through a central manager as both LRC and AURC protocols do. Although a barrier may sometimes reveal a true imbalance in computation (see LU, Figure 10), in other cases, such as

Ocean, the imbalance may occur as a side effect of the large and non-uniform costs of page misses in an SVM system. If some processor suffers a disproportionate number of page faults, this will delay its reaching the barrier, thus increasing the cost of synchronization for all processors.

The overhead of passing a lock between two nodes is always smaller for AURC than LRC, because AURC uses simpler data structures. On the other hand, increased network traffic can artificially slow down the lock path, sometimes making the cost of synchronization a little higher in AURC (Water).

6 Related Work

The concept of shared virtual memory (also called distributed shared memory) was proposed in Li's Ph.D. thesis in 1986 [17, 19]. It was first implemented on a network of workstations [18] and then applied to large-scale multicomputer systems [20, 19, 21, 29].

Relaxed consistency models [9] allow shared virtual memory to reduce the cost of false sharing. Recent research in this area includes [5, 6, 14, 1, 23, 24, 14, 7]. New coherency protocols improving the performance of release consistency include lazy release consistency [14], and entry consistency [1], in which all shared data is explicitly associated with some synchronization variable.

The TreadMarks library [13] is an example of state-of-the-art implementations of shared virtual memory on stock hardware. It uses lazy release consistency and allows for multiple writers. This implementation provides respectable performance in the absence of fine-grain sharing.

Software-only techniques to control fine-grain accesses to shared memory have been proposed. Blizzard-S [25] rewrites an existing executable file to insert a state table lookup before every shared-memory reference. This technique works well in the presence of fine-grain sharing; for well-structured programs it imposes substantial overhead compared to more traditional shared virtual memory implementations.

The PLUS [2], Galactica Net [12], Merlin [22] and its successor SESAME [27], systems implement hardware-based shared memory using a sort of write-through mechanism which is similar in some ways to automatic update. These systems do more in hardware, and thus are more expensive and complicated to build. Our automatic update mechanism propagates writes to only a single destination node; both PLUS and Galactica Net propagate updates along a "forwarding chain" of nodes. Although this sounds like a simple change, hardware forwarding of packets leads to a potential deadlock condition, since conventional multicomputer networks are deadlock-free only under the assumption that every node is willing to accept an arbitrary number of incoming packets even while its outgoing network link is blocked. PLUS and Galactica Net both avoid this deadlock by using deep FIFO buffers, and limiting the number of updates which each node may have "in flight" at a time. (The limit is eight in PLUS, five in Galactica Net.) Our hardware is simpler, and application performance is better, because we do not have to enforce such a limit.

Memory Channel [10] network allows remote memory to be mapped into the local virtual address space but without a corresponding local memory mapping. This is why writes to remote memory are not automatically performed locally at the same virtual address, making a software shared-memory scheme more difficult to implement.

7 Conclusions

This paper proposed three release consistency protocols Copyset-2, Copyset-N, and AURC for shared virtual memory system implementations, in order to take advantage of the automatic update communication mechanism in virtual memory mapped network interfaces. We have shown that the hybrid protocol AURC performs the best. Our simulation results of several Splash-2 benchmark applications show that on a network of up to 16 processors, AURC consistently outperforms LRC. For 16 processors, AURC improved the speedups up to 2.5 times, with the average speedup improved from 5.9 to over 8.3.

We learned several important things from our performance measurements:

- The AURC protocol suffers fewer page faults than the LRC protocol. The AURC protocol has no page faults on any of the two nodes for for pages shared by two nodes, and has no page faults on the owner node for pages shared by more than two nodes, whereas the LRC method has page faults for both cases.

- The AURC protocol pays no software overhead for calculating updates to shared pages. It takes advantage of the automatic update mechanism to accomplish the update propagation in hardware, whereas the LRC method pays significant software overhead in making page copies and computing diffs.

- Although the AURC method causes more network traffic than LRC (within 30% for our applications), AURC generates less contention on the local memory bus than LRC for our simulated architecture.

- The overhead of passing locks between two nodes is often smaller for AURC than LRC because AURC uses simpler data structures. However, sometimes increased network traffic can slow down the lock path in AURC.

In summary, we expect that the availability of automatic update can increase the range of applications for which shared virtual memory is feasible.

Many questions are still open. We have not compared the performance of the AURC approach with the hardware-only approach where directory-based cache coherence is implemented in hardware. We expect that for grid applications, the performance difference between the two approaches is small. Another interesting question is the performance implications of faster I/O bus bandwidths. Our preliminary data indicates that AURC benefits more than LRC when the communication bandwidth increases. We are currently conducting more experiments to answer these questions.

Acknowledgements

We are grateful to the anonymous referees whose comments helped us to improve the quality of this paper. Matthias Blumrich designed Shrimp network interface. We also extend our thanks to Jaswinder Pal Singh, Margaret Martonosi and Jim Philbin who offered us their generous assistance during this research. This project is sponsored in part by ARPA under grant N00014-91-J-4039 and N00014-95-1-1144, by NSF under grant MIP-9420653, and by Intel Scalable Systems Division. Ed Felten is supported by NSF National Young Investigator Award.

References

[1] 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.

[2] R. Bisiani and M. Ravishankar. Plus: A distributed shared-memory system. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 115–124, May 1990.

[3] M. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina. Virtual memory mapped network interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[4] 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, April 1994.

[5] L. Borrmann and M. Herdieckerhoff. A coherency model for virtually shared memory. In *1990 International Conference on Parallel Processing*, August 1990.

[6] 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.

[7] A.L. 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, April 1994.

[8] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 144–155, May 1993.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[10] Richard Gillett. Memory channel network for pci. In *Proceedings of Hot Interconnects '95 Symposium*, August 1995.

[11] S.A. Herrod. *TangoLite; A Multiprocessor Simulation Environment.* Computer Systems Laboratory, Stanford University, 1994.

[12] Andrew W. Wilson Jr. Richard P. LaRowe Jr. and Marc J. Teller. Hardware assist for distributed shared memory. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 246–255, May 1993.

[13] 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.

[14] 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.

[15] L.I. Kontothanassis and M.L. Scott. Software cache coherence for large scale multiprocessors. In *Proceedings of the First International Symposium on High Performance Computer Architecture*, January 1995.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[17] K. Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors.* PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[18] 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.

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

[20] K. Li and R. Schaefer. A hypercube shared virtual memory. In *Proceedings of the 1989 International Parallel Processing Conference*, volume Vol:I Architecture, pages 125–132, August 1989.

[21] Kai Li. Scalability issues on of shared virtual memory for multicomputers. In M. Dubois and S.S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 263–280. Kluwer Academic Publishers, May 1992.

[22] Creve Maples. A high-performance, memory-based interconnection system for multicomputer environments. In *Proceedings of Supercomputing '90*, pages 295–304, November 1990.

[23] K. Petersen and K. Li. Cache coherence for shared memory multiprocessors based on virtual memory support. In *Proceedings of the IEEE 7th International Parallel Processing Symposium*, April 1993.

[24] K. Petersen and K. Li. An evaluation of multiprocessor cache coherence based on virtual memory support. In *Proceedings of the IEEE 8th International Parallel Processing Symposium*, April 1994.

[25] 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.

[26] Ch. A. Thekkath and H.M. Levy. Hardware and software support for efficient exception handling. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–121, October 1994.

[27] Larry D. Wittie, Gudjon Hermannsson, and Ai Li. Eager sharing for efficient massive parallelism. In *Proceedings of the 1992 International Conference on Parall el Processing*, pages 251–255, August 1992.

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

[29] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory: An experimental study. *IEEE Transactions on Parallel and Distributed Computing*, 3(5):540–554, May 1992.