Shared Virtual Memory with Automatic Update Support

Liviu Iftode
Department of Computer Science
Rutgers University
Piscataway, NJ 08855
iftode@cs.rutgers.edu

Matthias Blumrich, Cezary Dubnicki, David L. Oppenheimer, Jaswinder Pal Singh and Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

Shared virtual memory systems provide the abstraction of a shared address space on top of a message-passing communication architecture. The overall performance of an SVM system therefore depends on both the raw performance of the underlying communication mechanism and the efficiency with which the SVM protocol uses that mechanism. The Automatic Update Release Consistency (AURC) protocol was proposed to take advantage of simple memory-mapped communication and automatic update support to accelerate a shared virtual memory protocol. However, there has not yet been a real system on which an implementation of this protocol could be evaluated.

This paper reports our evaluation of AURC on the *SHRIMP* multicomputer, the only hardware platform that supports an automatic update mechanism. Automatic update propagates local memory writes to remote memory locations automatically. We compare the AURC protocol with its all-software counterpart protocol, Home-based Lazy Release Consistency (HLRC). We first show that virtual memory-mapped communication itself helps all home-based LRC protocols because these protocols can easily take advantage of the zero-copy communication allowed by the memory mapping mechanism. By integrating AU support into the protocol as well, the AURC protocol can further improve performance. For applications with write-write false sharing, an AU-based multiple-writer protocol can significantly outperform an all-software home-based multiple-writer LRC protocol that uses diffs. For applications without much write-write false sharing, the two protocols perform similarly. Our results also show that write-through caching and automatic update traffic does not perturb the computation, validating the implementation as achieving its goals.

1 Introduction

Shared virtual memory systems provide the abstraction of a shared address space on top of a message-passing communication architecture. The overall performance of an SVM system therefore depends on both the raw performance of the underlying communication mechanism and the efficiency with which the SVM protocol uses that mechanism. In recent years, industrial and academic researchers have focused a great deal of work on the interconnect and messaging layer so that the performance of these components might improve at a rate comparable to that of processor speed. Memory-mapped network interfaces such as the SHRIMP [6] network interface and Digital's Memory Channel [12] transfer individual memory updates from one virtual memory to another across the network in less than 4 microseconds in a non-blocking fashion. An interesting question is how well shared virtual memory protocols can take advantage of such network interface support.

A number of shared virtual memory protocols have been proposed that use relaxed consistency models to tolerate communication latency and end-point overhead but most of these assume a standard network interface. Munin [7], TreadMarks [20] and Home-based Lazy Release Consistency (HLRC) [28], for example, all implement multiple writer protocols without special network interface support. These protocols use the CPU to compute local updates by determining the difference (*diff*) between a clean and dirty copy of the page that has been written. The protocols differ in how and when they propagate and merge diffs.

Recently, several protocols have been proposed to take advantage of memory-mapped communication that supports fine-grained remote writes [13, 15, 22]. These protocols are all home-based Lazy Release Consistency

(HLRC) protocols in which each page is assigned a fixed node called *home* to collect updates from multiple writers. The Automatic Update Release Consistency (AURC) protocol was the first proposal to take advantage of memory-mapped communication to implement an LRC protocol [13]. It uses memory-mapped communication in two ways. First, it maps non-home copies of a shared page to the home page. Once these mappings are established, the network interface ensures that writes performed to the local copy of the page are transparently propagated to the home copy by the Automatic Update (AU) mechanism. Second, it uses memory-mapped communication to transfer data from the home page to a non-home page directly to bring the non-home copy up-to-date. Such a data transfer requires very little software overhead, no memory copy, and no additional overhead at the receiving node.

A previous paper compares the AURC protocol with a distributed diff-based LRC protocol [14] using a simulator, but its evaluation has two limitations from the viewpoint of evaluating memory mapped communication and AU support. First, the protocols it compares differ not only in the update propagation mechanism used (diffs versus hardware-supported automatic update), but also in the type of multiple-writer scheme used (distributed versus home-based). As a result, the contribution of the AU support to the improvement in performance cannot be isolated from the contribution of the different scheme used for merging updates from multiple writers. Second, because the study was based on simulations, it was not possible to accurately account for the effects of network traffic, or to capture any possible limitations in the AU mechanism that might be visible only in a real implementation, such as its interaction with other architectural features. AURC could generate more network traffic of some kinds than all-software LRC because AURC propagates writes into the network in a write-through manner. For these reasons, a comparison of HLRC to AURC using real hardware is needed in order to truly assess the performance benefit of AU support for an SVM protocol.

With the *SHRIMP* system now running in a sizeable configuration, we now have an opportunity to assess the benefits for SVM protocols of a true AU mechanism implemented in hardware. Simulation studies have helped us understand what components of execution time and protocol activity AU can potentially help and to what extent, and what application characteristics make AU potentially valuable. Using our completed system, we can examine not only the benefits of AU but also how well its implementation in *SHRIMP* delivers on the potential capabilities of an AU mechanism, and what if any critical implementation bottlenecks remain.

We ran eight SPLASH-2 applications on top of several shared virtual memory protocols including AURC, HLRC, and AURC with Copyset-2 optimization (AURC-2). Our results show that virtual memory-mapped communication helps all home-based LRC protocols because these protocols can easily take advantage of the memory-mapping mechanism to reduce communication bandwidth. For applications with write-write false sharing, an AU-based multiple-writer protocol can significantly outperform an all-software home-based multiple-writer LRC protocol that uses diffs. For applications without much write-write false sharing, the two protocols perform similarly. In our experiments, we did not see contention in the AURC case, though it requires all shared, non-home pages to use a write-through caching strategy, while the HLRC protocol can employ a write-back caching strategy for all pages.

Notes to the reviewers: we will provide data on 16-node system and AURC with Copyset-2 extension in the final paper.

2 *SHRIMP* System

The *SHRIMP* multicomputer system consists of 16 Pentium PC nodes connected by an Intel Paragon routing network [26, 16]. Each PC uses an Intel Pentium Xpress motherboard [17] that holds a 66 Mhz Pentium CPU, 256 Kbytes of L2 cache, and 40 Mbytes of DRAM memory. Peripherals are connected to the system through the EISA expansion bus [3]. Main memory data can be cached by the CPU as write-through or write-back on a per-virtual-page basis, as specified in the process page tables. The caches snoop DMA transactions and automatically invalidate the corresponding cache lines, thereby keeping the caches consistent with *all* main memory updates, including those from EISA bus masters.

The custom network interface is the key system component of the *SHRIMP* system: it connects each PC node to the routing backplane and implements hardware support for virtual memory-mapped communication (VMMC) [6]. The single-word update latency from one node's virtual memory to that of another node is under 4 microseconds. The data transfer bandwidth achieved is about 25 Mbytes/sec, which is close to the PC's I/O bus bandwidth.

The network interface supports the VMMC abstraction in three ways. First, it provides an automatic update mechanism that allows memory writes snooped from the Xpress memory bus to be propagated to the physical memory of a remote node, according to an import-export mapping established by the sender and receiver. Because it does not require an explicit send instruction, AU allows data to be sent with no software overhead on the sending side. Second, the network interface supports *deliberate update* (DU) operations. An explicit send instruction is required for DU, but a user-level DMA mechanism [6] reduces the send overhead to a few instructions. Third, the network interface supports a notification mechanism that uses fast interrupts to notify a receiving process that data has arrived (if the receiving process has requested notifications for a particular mapping).

The shared virtual memory systems described in this paper are built on top of the VMMC library, a thin user-level library that implements the VMMC API. This library implements calls for exporting and importing memory buffers, sending data with and without notifications, and managing receive buffer memory.

3 Protocols

This section describes three coherence protocols we have compared on the *SHRIMP* multicomputer system: Home-base Lazy Release Consistency (HLRC) [28], Automatic Update Release Consistency (AURC) [13], and AURC with Copyset-2 optimization (AURC-2) [13] We will first briefly review the lazy release consistency model.

3.1 Lazy Release Consistency

Lazy Release Consistency (LRC) is a consistency model similar to release consistency [11] that is intended for software implementation. It delays the propagation of (page) invalidations until the latest possible acquire time. To achieve memory coherence with such a degree of laziness, the LRC protocol uses vector timestamps to maintain the "happens-before" partial causal ordering between synchronization events. A timestamp is a logical counter of the synchronization events that occur at a node. A vector timestamp indicates the latest synchronization event that a node has seen (*i.e.* is aware of) from every other node.

When a node acquires a lock, the node sends its *timestamp vector* with its lock request message. The node that last released the lock uses the vector timestamp received with the lock request to determine the differences between the set of intervals seen by itself (available from its own timestamp vector) and the set of intervals seen by the acquirer. The differences are sent back with the lock reply as "write notices" which are lists of page invalidations, one for each interval. Thus a write notice that a requester receives for an interval contains the identities of the pages the requester needs to invalidate, corresponding to that interval (which it has not yet seen). In this way an LRC protocol implements causal ordering between the events performed before the release and the events that occur after the acquire, thus satisfying the RC memory model.

LRC was first proposed, implemented, and evaluated in the TreadMarks SVM system [20]. TreadMarks supports concurrent writers to the same page using a software multiple-writer scheme based on distributed diffs. Every writer records locally the changes it makes to every shared page during each interval. These records, called *diffs*, are computed by comparing the current dirty copy of the page against a clean copy, called a *twin*, which was created before the first write occurring in that interval. When a processor page faults, it makes its local copy of a page consistent by obtaining the diffs from all the "last" writers to that page (according to the vector timestamps described earlier) and applying those diffs to its local copy in the proper causal order. Diffs can be created eagerly at the end of intervals or lazily when the first request for a diff arrives from another node.

3.2 Home-Based LRC Protocols

The lazy release consistency protocol based on distributed diffs was a major advance in SVM, but it has some drawbacks. The first problem is that a node may have to obtain diffs from several nodes when there is substantial write-write false sharing–precisely the case in which the multiple writer mechanism is useful. This obtaining of diffs requires several expensive round-trip messages upon a page fault. In the worst case, $O(n^2)$ messages must be exchanged. This happens when all nodes are last writers and all need the page, e.g. all nodes false write-share the page before the barrier and then read the page after the barrier. The best case is when the sharing pattern is migratory; then diffs are totally ordered and can be fetched in one message from the last writer. Without using an adaptive scheme of the type recently proposed [2, 19], the sizes of the messages increases during the migration due to diff accumulation (since a processor is fetching diffs, it fetches not only the diffs created by the last writer but also those created by previous writers in the intervals that it has not already seen).

Another problem with distributed diffs is that the same diff may need to be applied as many as $O(n)$ times at $O(n)$ different nodes that fetch that diff. Diff application and diff creation incur a lot of cache misses, so diff application should be minimized.

Finally, since the node that creates a diff stores it locally until no other node may need it (which is difficult to predict), distributed diff schemes usually do not discard the diffs until they are garbage collected. This can result in significant memory consumption by the protocol during program execution; indeed, the memory required to store diffs can eventually exceed the memory used by the application [28]. Garbage collection is usually triggered at global barriers to free up the diff storage, but this garbage collection is itself an expensive global operation that increases protocol overhead.

A home-based multiple-writer scheme is an alternative approach to supporting multiple concurrent writers. This approach combines eager propagation of data at release time with lazy invalidations. For every shared page, a single node (usually the first writer) is designated as the page's *home*. The home's copy of the page is eagerly updated before the next release point. Non-home copies are updated on demand at page fault time by fetching the whole page from the home, rather than as diffs from the recent writers. Non-home copies are invalidated on acquire according to the standard LRC algorithm described earlier.

One way to propagate writes to a page's home is to compute diffs at the next synchronization point and to then send them explicitly. This method can be used to implement a home-based protocol entirely in software. Another way to propagate these writes is to use the automatic update hardware mechanism provided by a network interface like SHRIMP [6] or Memory Channel [12].

A previous study showed that home-based protocols could achieve better performance and scalability than the traditional LRC protocol based on distributed diffs [28]. There are several reasons for this: reads and writes to a page by the page's home node can always be performed using local operations without page faults or communication, diffs only need to be applied once (at the page's home), non-home nodes update their copy of a page using a single round-trip message (to the home) rather than potentially several messages, memory consumption due to protocol data is low The main drawback of home-based protocols is the need to select homes for pages well, i.e. to distribute data carefully across nodes; of course, this may also be needed in hardware-coherent CC-NUMA machines.

Home-based protocols are well-suited to the current generation of memory-mapped network interfaces because a page can be transferred directly into the physical page to which a faulting page is mapped, with zero-copy and no additional overhead. Moreover, given that processor speed is growing faster than memory access time (making diff computation and application more expensive relative to program execution, especially for programs that operate primarily out of their cache) and communication bandwidth is growing faster than message latency is shrinking, the performance gap between the home-based approach and the traditional distributed diff approach may increase in the future. This trend motivates us to compare all-software home-based protocols with those that take advantage of memory-mapped network interface features.

3.2.1 Home-based LRC

The all-software HLRC uses the same diff-based mechanism as traditional LRC to detect updates. But unlike traditional LRC, in which diffs are created on demand, stored and provided possibly many times, in HLRC diffs are created at the synchronization time, immediately sent to the home, then discarded. At the home diffs are applied in the arrival order and then discarded.

Version numbers represented as per-shared-page timestamp vectors are used to maintain the causal order of updates. The page version at the home is incremented each time a received diff is applied. Non-home nodes also keep a vector timestamp per page, indicating the current version of valid pages or the expected version of invalid ones. On a page fault the faulting node requests the expected version of the page from the home. The home sends the page when that version is available. Because the home sends back its timestamp vector along with the page, a prefetching effect can be obtained and future invalidations may be avoided.

One way to utilize hardware AU support in an HLRC protocol is to simply use the AU mechanism to map the page holding the diffs to the home page. As the protocol computes the diffs at a release point, the memory writes that create the diffs are be propagated to the home page automatically. This method allows the protocol to avoid buffering diffs and sending them to the home in an explicit message. We shall evaluate this approach (HLRC-AU) in this paper. A more promising approach is to incorporate the AU support into the protocol itself.

3.2.2 Automatic-Update RC

Compared to the traditional LRC protocol, the software HLRC protocol can reduce communication traffic and memory overhead in the presence of write-write false sharing. But HLRC does not eliminate the expensive operations related to manipulating diffs (twin creation, diff creation, and diff application); these operations can add significant memory and time overhead to the HLRC protocol.

Automatic Update Release Consistency (AURC) implements a home-based multiple-writer scheme using automatic update (AU) hardware instead of software diffs to detect, propagate to the home page, and merge writes performed by concurrent writers. This method eliminates diffs entirely from the protocol (see Table 3). The basic idea of AURC is to establish an AU mapping from the non-home copies of a shared page to the home copy, so that local writes to these copies at the non-home nodes are automatically propagated to the home. The rest of the protocol is similar to HLRC. The main difference between AURC and HLRC is that AURC propagates writes immediately, through the AU mappings, while HLRC detects modified words and propagates them at the next release time. AURC will therefore propagate multiple updates to the same memory location multiple times, whereas HLRC will propagate only the last change at the next release time. On the other hand, AU update propagation is pipelined, whereas diff computations and propagations in HLRC can be bursty.

A minor difference between AURC and HLRC is that in AURC timestamps are used not only to ensure causal ordering before synchronization events, as in HLRC, but also to flush the AU links at synchronization time. Because the network interface performs write combining in the hope of reducing the need to send frequent small messages, a block of writes may be propagated to the destination (home) by the network interface after an arbitrary time delay. Flushing an AU link forces the writes in the sender's write buffer or network queue into the network and therefore to the receiver; this flush is accomplished by sending a time-stamp. The flush operation, combined with the fact that AU messages are delivered in FIFO order, ensures that once the timestamp sent by the writer arrives in the corresponding slot of the home's timestamp vector, all the previous writes performed by that writer are guaranteed to be in place in the home's copy. Because AURC's relaxed consistency model tolerates lazy updating, AURC can take advantage of the performance benefit of the network interfaces write-combining feature.

The relative performance of AURC and HLRC, representing a high-level trade-off between potentially higher bandwidth needs in AURC (due to multiple writes to the same word being propagated multiple times) and higher end-point overhead (and perhaps end-point contention) in HLRC due to diff computation and bursty propagation, has not been evaluated using an actual implementation because no AU implementation of this type has existed until now. Previous comparisons between AURC and software-based protocols were performed using simulations [13, 14], making this the first opportunity to evaluate the actual performance benefit of AURC for SVM on a real *SHRIMP* system.

3.2.3 Automatic-Update RC with Copyset-2

Previous research in hardware support for update-based shared memory protocols has suggested multicast support for a full-fledged update-based coherence scheme, with updates propagated in circular rings [18, 5]. As the *SHRIMP* hardware design philosophy emphasizes simplicity, we ruled out complicated features such as rings and multicast. Thus, the AU hardware can only propagate a write to one destination, and therefore lends itself to a home-based protocol in which writes from potentially multiple writers are propagated by mapping those writers one-to-one to a single home and merging the writes at the home. Full-page fetches from the home are performed through explicit messages rather than using AU.

The standard AURC scheme maps non-home copies of a shared page to the home copy, but does not map the home copy out to any other node. Even without multicast an opportunity exists to map the home to one other copy using the one-to-one mapping support. The copyset-2 optimization is an extension to the AURC protocol (AURC-2) that takes advantage of this fact that the home node can also propagate its updates to a non-home page. Of course the non-home node must still fetch the page from the home in order to receive updates from other non-home nodes.

The most valuable use of this protocol is for pages that are shared by only two nodes, one of which is the home. In this case AURC-2 can keep the two copies consistent through the AU communication (the non-home page's writes are propagated to the home, and the home's to it, so there are no page faults at all.) Flushing is still needed to guarantee the completion of the remote updates, but no full-page transfers are needed after the initial page transfer.

The AURC-2 protocol is particularly useful for applications that exhibit nearest-neighbor communication [13], but it can reduce the total number of page faults for other applications as well since there are often some pages that are shared by only two nodes in programs that do not exhibit a nearest-neighbor communication pattern. The possible drawback of this method, compared to standard AURC, is that AURC-2 increases the amount of overall network traffic since the home nodes also generate AU write traffic.

4 Applications

We evaluated the performance of our SVM protocols using a subset of the SPLASH-2 applications and kernels [25] that exhibit both regular and irregular sharing patterns.

Barnes uses the Barnes-Hut hierarchical N-body method to simulate the interactions among a system of particles over time. The computational domain is represented as an octree of space cells. The leaves of the octree contain particles, and the particles and space cells are distributed to processors based on their positions in space. At each time step of the simulation the octree is rebuilt based on the current positions of the bodies, and each processor partially traverses the tree to compute the forces on the particles assigned to it. The interaction between a particle and a group of distant particles is computed as the force between the particle and the center of mass of the distant particles; the tree structure allows the latter to be determined easily. At the end of each time step the positions of the bodies are updated based on the computed forces. The properties of particles and cells are updated in an order that has little to do with their placement in the particle and cell arrays, leading to false sharing at page granularity. This false sharing is particularly problematic in the tree-building phase because many of the updates in that phase are protected by per-cell locks.

While the calculation of forces is by far the dominant component of Barnes's execution time on hardware CC-NUMA systems, on SVM systems the time required to calculate the forces is approximately equal to that required to rebuild the tree because SVM's higher communication and synchronization costs drive up the time required to rebuild the tree.

FFT is a high-performance FFT kernel. The communication in an n-point FFT is essentially a matrix transposition of a $\sqrt{n}$ by $\sqrt{n}$ matrix. FFT distributes matrices so that every processor is assigned a contiguous set of n/p rows, and the source and destination matrices are reversed for every transpose. In this version of FFT, each processor in a transpose reads an $\frac{\sqrt{n}}{p}$ by $\frac{\sqrt{n}}{p}$ submatrix from every other processor and writes it into its local partition or set of rows. Consequently the write access granularity is coarse while the read access granularity depends on n and p (a processor reads subrows of size $\frac{\sqrt{n}}{p}$ from the other processors). For typical values of these parameters, the read granularity is often moderate, leading to fragmentation but not false sharing. Due to the extra communication caused by the fragmentation effect, as well as the relatively high communication to computation ratio, the performance of FFT on a page-based software shared memory system is generally modest unless the problem size is very large, which decreases fragmentation.

LU is a kernel that 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, we used a version with an optimized data structure for the matrix, namely a 4-d array, in which each 16x16 element block is allocated contiguously in virtual memory and the blocks assigned to a processor are also allocated contiguously. Barriers are used to ensure synchronization between the processor producing the pivot row blocks and the processors consuming them, as well as between outermost loop iterations.

Ocean is a fluid dynamics application that simulates large-scale ocean movements. Spatial partial differential equations are solved at each time-step of the program using a restricted Red-Black Gauss-Seidel Multigrid solver. Ocean is representative of the class of problems that are solved using regular grid near-neighbor computations. At each iteration the computation performed on each element of the grid requires the values of its four neighbors. Work is assigned to processors by statically splitting the grid and assigning a partition to each processor. Nearest-neighbor communication occurs between processors assigned adjacent blocks of the grid. We ran two versions of ocean with two different partitioning methods: ocean-square uses square sub-grids while ocean-row partitions the matrix in blocks of n/p contiguous full rows. Although the inherent communication to computation ratio is smaller when square-shaped sub-grids are used, the communication induced by the page-based granularity of the SVM protocol makes the overall communication to computation ratio worse for ocean-square than for ocean-row due to fragmentation and false sharing at the columnar partition boundaries.

Radix sorts a series of integer keys into ascending order. The dominant phase of Radix is the key permutation phase. In Radix a processor reads its locally-allocated n/p contiguous keys from a source array and writes them to a destination array using a highly scattered and irregular permutation. For a uniform distribution of key values, a processor writes contiguous sets of $\frac{n}{r*p}$ keys in the destination array (where r is the radix used); the r sets a processor writes are themselves separated by $p-1$ other such sets, and a processor's writes to its different sets are temporally interleaved in an unpredictable way. This write pattern induces substantial false-sharing at page granularity.

Water-Nsquared simulates a system of water molecules in liquid state using an $O(n^2)$ brute force method with a cutoff radius. The water molecules are allocated contiguously in an array of n molecules, and are partitioned among processors in contiguous pieces of n/p molecules each. Updates are accumulated locally between iterations and are performed once at the end of each iteration. In this latter phase, a processor updates its own n/p molecules and the following $(n/2 - n/p)$ molecules of other processors in the array using per-partition locks to protect the updates. Water-Nsquared exhibits a migratory sharing pattern in the updates but incurs little or no false sharing or fragmentation if the array partition size is a multiple of the page size and the partitions are page-aligned.

Water-Spatial solves the same problem as Water-Nsquared but using a spatial directory rather than a brute-force $O(n^2)$ method; this technique is particularly beneficial for large problems. The 3-d physical space is broken up into cells, and every processor is assigned a contiguous cubical partition of cells together with the linked lists of molecules in those cells. A processor reads data from the processors that own cells on the boundary of its partition, and may write the linked lists of these processors if molecules move across partitions. Migration of molecules between cells requires locking of linked lists, but this migration is infrequent and therefore has little impact on performance.

Application	Problem Size	Sequential Execution Time (secs)
Barnes	8K bodies	55
FFT	256K	23
LU	512×512	26
Ocean	130×130	13
Radix	1 M keys	7
Water-Nsquared	512 molecules	64
Water-Spatial	512 molecules	119

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

Table 1 shows the problem sizes and the sequential execution time for all applications. Problem sizes are small due to the current limitations of the memory mapping size in *SHRIMP*.

5 Performance Evaluation

5.1 Basic Costs

Table 2 shows the costs of the basic operations on the *SHRIMP* system.

Operation	Time in microseconds
Message Latency	6
4K Page Transfer	180
Message Notification	53
4K Page Copy	64
Page Fault	20
Page Protection	22

Table 2: Basic operations on *SHRIMP*

5.2 Results

Our performance evaluation consists of three parts. First, we examine the impact of using memory-mapped communication itself, which allows pages to be transferred without the copy-on-receive required in SVM implementations based on sockets or other standard message passing libraries.

Second, we evaluate the impact of using automatic update support simply to accelerate the communication in an all-software protocol. In particular, we implemented an HLRC protocol that use AU to propagate diffs transparently as they are produced, rather than buffering them and sending them using a Deliberate Update (DU) message as in standard HLRC. This eliminates the memory overhead of buffering and sending the diff.

Third, we take full advantage of AU support at the protocol level. We implemented AURC which eliminates diffs altogether and propagates writes to shared data to the remote home immediately rather than waiting for the next release. We compare AURC with the diff-based HLRC.

The contribution of the AURC-2 protocol extension is not evaluated in the current version of the paper. This extension is expected to benefit situations in which pages have two sharers, for example the nearest neighbor communication in Ocean. We are in the process of evaluating it and the results will be provided in the final version of this paper.

5.2.1 Benefits of Zero-copy Communication

Figure 1 presents the speedups obtained on an eight-processor *SHRIMP* for our applications using two implementations of the all-software protocol: HLRC-copy and HLRC. The difference between these two implementations is in how a page fault is handled at the requester. In HLRC when a page is sent from the home in response to a page request, it is placed directly into the physical page frame for the local copy of that page, using memory mapped communication. HLRC-copy copies page for each page transfer as is done in SVM implementations based on sockets or some other message passing systems. In HLRC-copy the node receives the page in a special buffer then the node copies the page from the special buffer into the local copy of that page.

The HLRC protocol achieves speedups over uniprocessor execution in the range of 22%-84% parallel efficiency. The speedups are good for Water-Nsquared, Water-Spatial, LU, FFT and Ocean-row, and reasonable for Barnes given the small problem sizes we were able to run. Breakdowns of execution time show that compared to HLRC, HLRC-copy spends twice as much time in communication, indicating that the extra copy operation halves the effective page transfer bandwidth. This result is consistent with the results

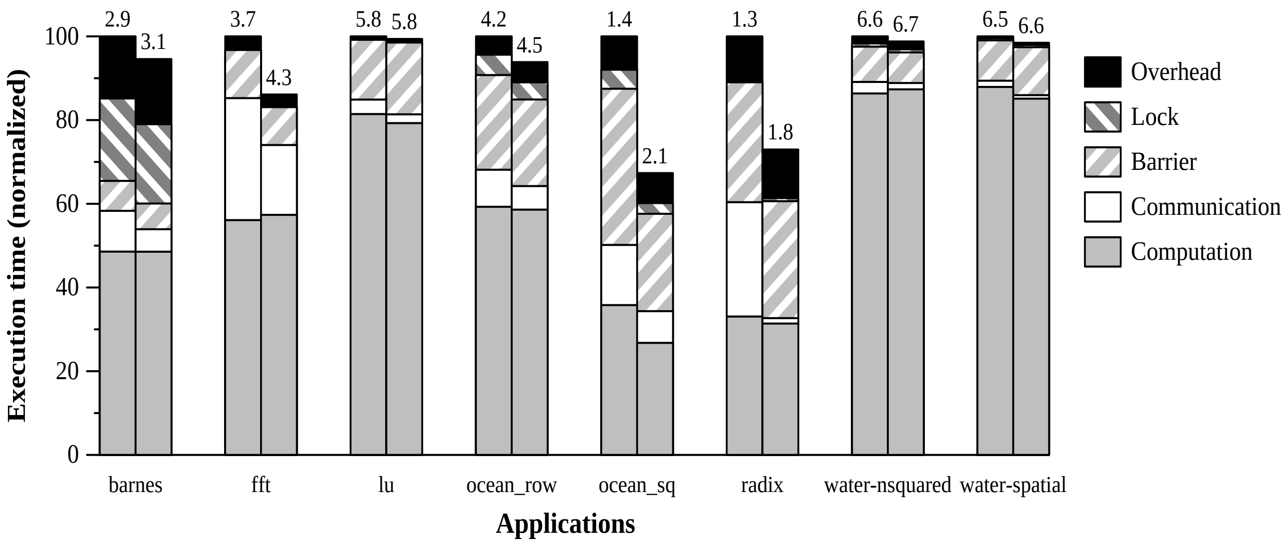

Figure 1: Comparative performance: HLRC-copy vs. HLRC. Left hand side of each pair of bars is HLRC-copy. Number on top of each bar is speedup.

reported in [10]. Overall, HLRC performs significantly better than HLRC-copy for the applications with high communication-to-computation ratios (FFT, Radix and Ocean-square), a little better for those with medium communication-to-computation ratios (Barnes and Ocean-row), and similarly for those with very low communication-to-computation ratios (Water and LU). We conclude from these results that a communication architecture that provides zero-copy data transfer, such as memory mapped communication, is very important to the efficiency of an SVM protocol.

5.2.2 AU as a Communication Accelerator

In contrast to the benefit of removing the copy, the further benefit of using AU to transfer diffs via fine-grained, overlapped writes (HLRC-AU) rather than buffering them together and sending them with DU in standard HLRC is very small.

Figure 2 show that HLRC-AU either slightly improves performance for some applications (e.g. Radix and Ocean-square) or it slightly hurts the performance of some others (e.g. Ocean-row, Barnes and Water-spatial). This is because while AU reduces overhead at the sender compared to explicit transfer, it increases the occupancy of the sender's and receiver's memory buses, possibly reducing the ability of nodes to respond quickly to remote requests. Nonetheless, the performance effects of this natural use of AU are small.

5.2.3 Using AU to Implement AURC

Figure 3 shows the performance comparison between HLRC and AURC. The results confirm our expectations and hopes for AURC.

AURC significantly outperforms HLRC for applications that exhibit significant write-write false sharing: by 15%, 20%, 52%, and 78% for Ocean-row, Barnes, Ocean-square, and Radix, respectively. The communication in Water-Nsquared and Water-Spatial also benefits from AURC, but overall performance does not benefit significantly due to low communication-to-computation ratio in these applications. Radix, an application known for its poor performance on SVM systems [14] obtains a substantial performance improvement. This is because Radix performs irregularly scattered writes to remotely allocated data in its communication phase, which leads to a lot of diffing in HLRC but is quite well suited to the automatic fine-grained propagation of writes in AURC. Using AURC five applications improve their parallel efficiency compared to HLRC, exceeding 50% with the exception of Barnes, Ocean-square and Radix, which come between at 20% and 40%.

On the other hand, HLRC and AURC perform similarly for applications that exhibit little or no multiple-writer false sharing. These applications are single-writer applications even at page granularity, and if the pages that a processor writes are allocated in its local memory then there is no need for the propagation of updates to remote homes. Thus, there is no need for diffs in HLRC, and the AU mechanism is hardly used at all.

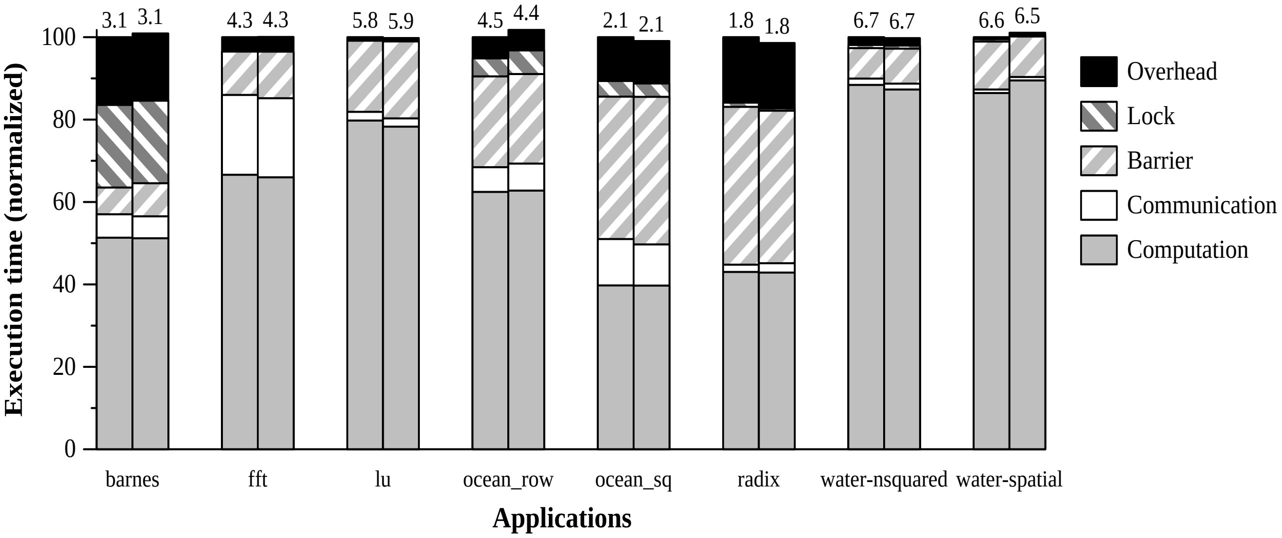

Figure 2: Comparative performance: HLRC vs HLRC-AU. Left hand side of each pair of bars is HLRC. Number on top of each bar is speedup.

Detailed analysis of execution time and protocol cost breakdowns shows that AURC does its job of eliminating diff related overhead very well (see Table 3), without noticeably increasing computation time or network contention due to write-through page mapping and AU traffic. This is true even in Radix which tends to stress communication bandwidth. Thus, it appears to deliver completely on the aspect of performance improvement for which it was designed. The effect on overall performance depends on the characteristics of the application and how much time is spent in computing diffs and its side-effects. The savings in protocol overhead (time the processor spends executing protocol code) is more than a factor of 4 in Radix, 3 in Barnes, and 2 in the two Ocean applications.

As a side effect of reducing protocol overhead, the synchronization time of both locks and barriers is also reduced. This is because the computation of diffs at release points dilates the critical sections protected by lock, and hence increases the synchronization and average wait time for a lock. This serialization and the diff computation itself may also magnify load imbalances among processors, causing more idle time to be spent waiting at barriers. AURC occasionally increases communication or synchronization time due to increased network traffic, but we can see that such costs associated with AURC are much smaller in applications than the benefits that AURC affords to other components of a program's execution time.

Table 4 shows that by replacing diffs with AU, AURC achieves significant savings in the number of explicit messages (up to an order of magnitude difference in the Radix case). In terms of data traffic the gain of AURC due to diff elimination is smaller than the gain in number of messages. The dominator of communication traffic in HLRC is full page transfer. In the current system the AU traffic cannot be measured. However, if we assume that the AU traffic in AURC is comparable to the diff traffic in HLRC then we can conclude that the AU traffic doesn't contribute much to the total traffic in AURC. The result implies that a good selection of homes (whose writes do not generate AU) is helpful. This agrees with the previous simulation results [13]. It is further confirmed by the absence of visible effects of contention in the computation time of AURC compared to HLRC.

6 Write-Only Pages

The Radix program captures an interesting effect of the coherence protocol, which is due to its inability to distinguish write-write false sharing from read-write false sharing. This inability stems from the fact that the virtual-memory hardware allows read-only but not *write-only* permission to be set in the TLB. If virtual-memory hardware supported write-only access permission then the SVM protocol could read-protect a page while still allowing writes. This would allow concurrent writers to write to their local copies without generating page misses until a read occurred.

In the absence of such a capability the SVM protocol can turn writers into involuntary readers, thus generating unnecessary page transfer traffic. In Radix this

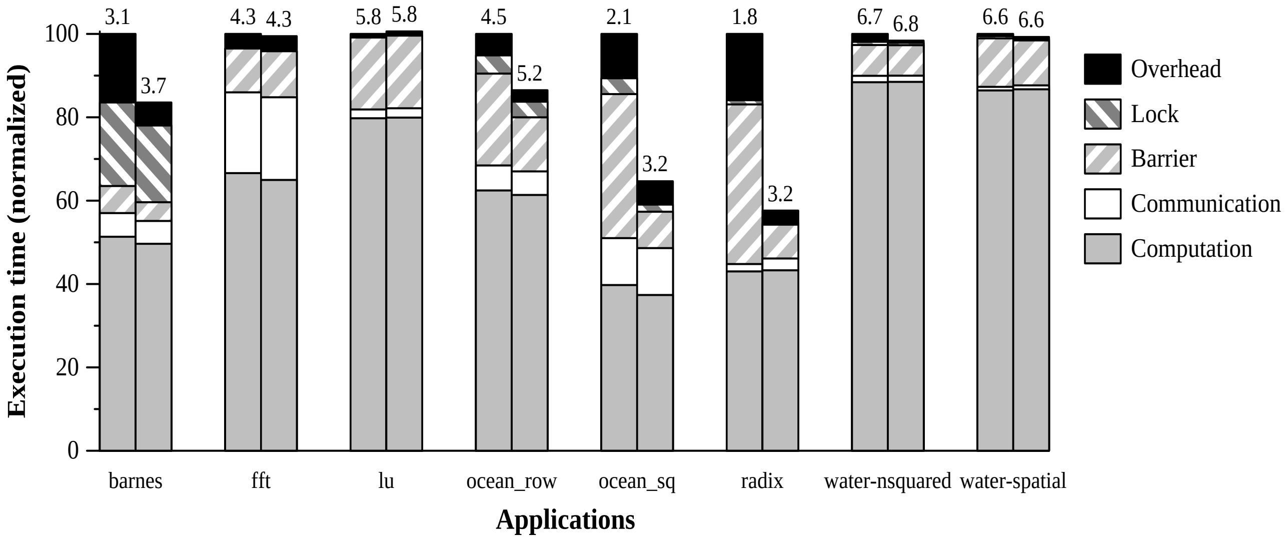

Figure 3: HLRC vs AURC. Left hand side of each pair of bars is HLRC. Number on top of each bar is speedup.

Application	Page misses		Diffs Created		Diffs Applied		Lock	Barriers
	HLRC	AURC	HLRC	AURC	HLRC	AURC	Acquires	
Barnes	2,517	2,498	3,398	0	3,398	0	20,468	8
FFT	2,240	2,240	0	0	0	0	1	10
LU	234	234	0	0	0	0	1	65
Ocean-row	432	443	365	0	365	0	92	399
Ocean-sq	1,822	1,740	1,785	0	1,785	0	92	399
Radix	166	166	1,941	0	1,941	0	25	10
Water-Nsquared	391	387	222	0	222	0	64	22
Water-Spatial	631	641	111	0	111	0	22	18

Table 3: Average number of operations on each node.

situation occurs at each step during the key permutation phase. The writer of the output array does not itself need to see the data that it writes into the remotely allocated portions of the array. We would like to label these pages of the output array for a permutation as write-only, so the writes to them do not cause page faults. To evaluate the effect of supporting write-only pages for HLRC and AURC, we introduced explicit annotations in the application to indicate write-only pages. The protocol trusts the application and doesn't invalidate write-only pages, thus emulating a write-only permission control.

Figure 4 shows the effect of this optimization on both HLRC and AURC protocols. In both cases when write-only annotations are used, the number of page misses is dramatically reduced from 166 to less than 10, which in turn considerably reduces communication overhead. But due to diff overhead and its impact on synchronization overhead which is larger in HLRC, the impact of write-only optimization is larger for AURC than for HLRC but overall it remains small.

7 Related Work

Since shared virtual memory was first proposed ten years ago [23], a lot of work has been done on it. The Release Consistency (RC) model was proposed in order to improve hardware cache coherence [11]. The model was used to implement shared virtual memory and reduce false sharing by allowing multiple writers [7]. Lazy Release Consistency (LRC) [21, 8, 1] further relaxed the RC protocol to reduce protocol overhead. TreadMarks [20] was the first SVM implementation using the LRC protocol on a network of stock computers. That implementation has achieved respectable performance on small-scale machines.

Cashmere [22] is an eager Release Consistent (RC)

Application	Update traffic				Protocol traffic			
	Number of messages		Message traffic (Mbytes)		Number of messages		Message traffic (Mbytes)	
	HLRC	AURC	HLRC	AURC	HLRC	AURC	HLRC	AURC
Barnes	5,915	2,498	10.4	10.1	39,600	32,785	2.2	2.16
FFT	2,240	2,240	9.2	9.2	4,600	4,600	0.4	0.4
LU	234	234	0.9	0.9	990	990	0.04	0.04
Ocean-row	797	443	2.0	1.4	6,782	6,063	0.15	0.13
Ocean-sq	3,607	1,740	9.1	7.0	12,130	8,478	0.4	0.32
Radix	2,107	166	2.1	0.7	4,471	589	0.15	0.09
Water-Nsquared	613	387	1.6	1.3	1,813	1,365	0.09	0.08
Water-Spatial	742	641	1.8	1.7	1,880	1,668	0.1	0.1

Table 4: Average communication traffic on each node.

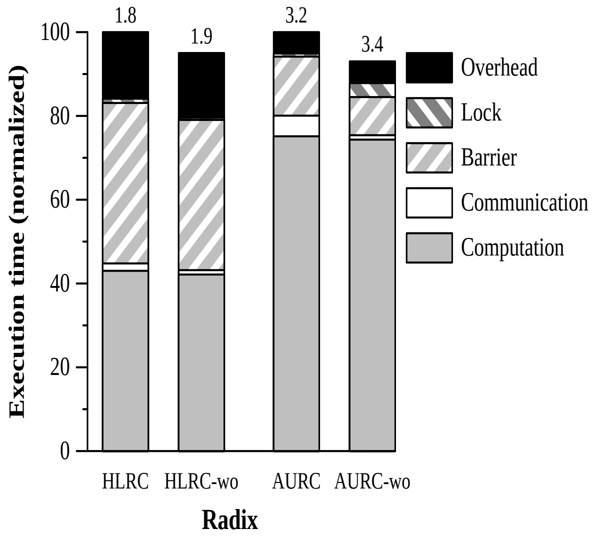

Figure 4: Effect of Write-Only pages on HLRC and AURC.

SVM protocol that implements a home-based multiple-writer scheme using the I/O remote write operations supported by the DEC Memory Channel network interface [12] rather than AU. Cashmere has been evaluated in comparison with TreadMarks [20], the best-known all-software distributed diff-based LRC protocol. However, the write propagation mechanism in Memory Channel requires explicit remote write instructions in software and therefore is not a transparent automatic update mechanism. Besides requiring explicit remote write operations to be inserted into the executable code, Memory Channels write propagation differs from that of AU in that it allows writes to be propagated selectively. Compared to AU, the explicit remote write operations might detract from the benefits of automatic write propagation, while the selective propagation might help performance by reducing network traffic. For these reasons the comparison between Cashmere and TreadMarks cannot conclusively assess the performance benefit of zero-software-overhead, non-selective AU support as provided in the *SHRIMP* network interface.

Bianchini et al. [4] proposed a dedicated protocol controller to offload some of the communication and coherence overheads from the computation processor. Using simulations they show that such a protocol processor can double the performance of TreadMarks on a 16-node configuration and that diff prefetching is not always beneficial.

The PLUS [5], Galactica Net [18], Merlin [24] 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 [12] 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.

In the context of compiler support for SVM systems research has been done in having the compiler detect write-only pages and prevent the SVM protocol from invalidating them at synchronization points [9].

8 Conclusions

We have investigated the performance benefit of automatic update in improving shared virtual memory protocols, on a real hardware implementation.

Using *SHRIMP*, the only available platform which supports automatic update exclusively in hardware, we implemented and evaluated several home-based protocols on a subset of SPLASH-2 applications. Our results confirmed the expectations set by earlier simulation studies.

First, we found that virtual memory-mapped communication helps all home-based LRC protocols because these protocols can easily take advantage of the zero-copy communication allowed by the memory mapping mechanism.

Second, we showed that by taking advantage of the automatic update mechanism in the LRC protocol, one can further improve the performance. For applications with write-write false sharing, an AU-based multiple-writer protocol can significantly outperform an all-software home-based multiple-writer LRC protocol that uses diffs. For applications without much write-write false sharing, the two protocols perform similarly.

Finally, in our experiments we did not see any effect on the execution time due to write-through caching and AU traffic. This says that the AU mechanism is indeed implemented in a way that accelerates shared virtual memory without any hidden costs, and that the expectations from AU in such protocols is confirmed.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementation. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[2] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *The 3rd IEEE Symposium on High-Performance Computer Architecture*, 1997.

[3] BCPR Services Inc. *EISA Specification, Version 3.12*, 1992.

[4] R. Bianchini, L.I Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory Systems. In *ASPLOS-VII*, 1996.

[10] E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[11] 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.

[12] Richard Gillett. Memory Channel Network for PCI. In *Proceedings of Hot Interconnects '95 Symposium*, August 1995.

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

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

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

[16] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[17] Intel Corporation. *Express Platforms Technical Product Summary: System Overview*, April 1993.

[18] 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.

[19] P. Keleher. Is There No Place Like Home. In *Submitted for publication*, 1997.

[20] 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.

[21] 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.

[22] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Ciernak, S. Parthasarathy, W. Meira Jr., S. Dwarkadas, and M. Scott. VM-based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *ISCA24*, 1997.

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

[24] Creve Maples. A High-Performance, Memory-Based Interconnection System For Multicomputer Environments. In *Proceedings of Supercomputing '90*, pages 295–304, November 1990.

[25] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, 1992. Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[26] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

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