TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems

Pete Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

TreadMarks is a *distributed shared memory* (DSM) system for standard Unix systems such as SunOS and Ultrix. This paper presents a performance evaluation of TreadMarks running on Ultrix using DECstation-5000/240's that are connected by a 100-Mbps switch-based ATM LAN and a 10-Mbps Ethernet. Our objective is to determine the efficiency of a user-level DSM implementation on commercially available workstations and operating systems.

We achieved good speedups on the 8-processor ATM network for Jacobi (7.4), TSP (7.2), Quicksort (6.3), and ILINK (5.7). For a slightly modified version of Water from the SPLASH benchmark suite, we achieved only moderate speedups (4.0) due to the high communication and synchronization rate. Speedups decline on the 10-Mbps Ethernet (5.5 for Jacobi, 6.5 for TSP, 4.2 for Quicksort, 5.1 for ILINK, and 2.1 for Water), reflecting the bandwidth limitations of the Ethernet. These results support the contention that, with suitable networking technology, DSM is a viable technique for parallel computation on clusters of workstations.

To achieve these speedups, TreadMarks goes to great lengths to reduce the amount of communication performed to maintain memory consistency. It uses a lazy implementation of release consistency, and it allows multiple concurrent writers to modify a page, reducing the impact of false sharing. Great care was taken to minimize communication overhead. In particular, on the ATM network, we used a standard low-level protocol, AAL3/4, bypassing the TCP/IP protocol stack. Unix communication overhead, however, remains the main obstacle in the way of better performance for programs like Water. Compared to the Unix communication overhead, memory management cost (both kernel and user level) is small and wire time is negligible.

1 Introduction

With increasing frequency, networks of workstations are being used as parallel computers. High-speed general-purpose networks and very powerful workstation processors have narrowed the performance gap between workstation clusters and supercomputers. Furthermore, the workstation approach provides a relatively low-cost, low-risk entry into the parallel computing arena. Many organizations already have an installed workstation base, no special hardware is required to use this facility as a parallel computer, and the resulting system can be easily maintained, extended and upgraded. We expect that the workstation cluster approach to parallel computing will gain further popularity, as advances in networking continue to improve its cost/performance ratio.

This research was supported in part by the National Science Foundation under Grants CCR-9116343, CCR-9211004, CDA-9222911, and CDA-9310073, by the Texas Advanced Technology Program under Grant 003604014, and by a NASA Graduate Fellowship.

Various software systems have been proposed and built to support parallel computation on workstation networks, e.g., tuple spaces [2], distributed shared memory [18], and message passing [23]. TreadMarks is a *distributed shared memory* (DSM) system [18]. DSM enables processes on different machines to share memory, even though the machines physically do not share memory (see Figure 1). This approach is attractive since most programmers find it easier to use than a message passing paradigm, which requires them to explicitly partition data and manage communication. With a global address space, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values.

Many DSM implementations have been reported in the literature (see [20] for an overview). Unfortunately, none of these implementations are widely available. Many run on in-house research platforms, rather than on generally available operating systems, or require kernel modifications that make them unappealing. Early DSM systems also suffered from performance problems. These early designs implemented the shared memory abstraction by imitating consistency protocols used by hardware shared memory multiprocessors. Given the large consistency units in DSM (virtual memory pages), false sharing was a serious problem for many applications.

TreadMarks overcomes most of these problems: it is an efficient DSM system that runs on commonly available Unix systems. This paper reports on an implementation on Ultrix using 8 DECStation-5000/240s, connected both by a 100-Mbps point-to-point ATM LAN and by a 10-Mbps Ethernet. The system has also been implemented on SunOS using SPARCstation-1's and -2's connected by a 10-Mbps Ethernet. The implementation is done at the *user* level, without modification to the operating system kernel. Furthermore, we do not rely on any particular compiler. Instead, our implementation relies on (user-level) memory management techniques to detect accesses and updates to shared data. In order to address the performance problems with earlier DSM systems, the TreadMarks implementation focuses on reducing the amount of communication necessary to keep the distributed memories consistent. It uses a lazy implementation [14] of release consistency [13] and multiple-writer protocols to reduce the impact of false sharing [8].

On the 100-Mbps ATM LAN, good speedups were achieved for Jacobi, TSP, Quicksort, and ILINK (a program from the genetic LINKAGE package [16]). TreadMarks achieved only a moderate speedup for a slightly modified version of the Water program from the SPLASH benchmark suite [22], because of the high synchronization and communication rates. We present a detailed decomposition of the overheads. For the applications measured, the software communication overhead is the bottleneck in achieving high performance for finer grained applications like Water. This is the case even when using a low-level adaptation layer protocol (AAL3/4) on the ATM network, bypassing the TCP/IP protocol stack. The communication overhead dominates the memory management and consistency overhead. On a 100-Mbps ATM LAN, the "wire" time is all but negligible.

The outline of the rest of this paper is as follows. Section 2 focuses on the principal design decisions: release consistency, lazy release consistency, multiple-writer protocols, and lazy diff creation. Section 3

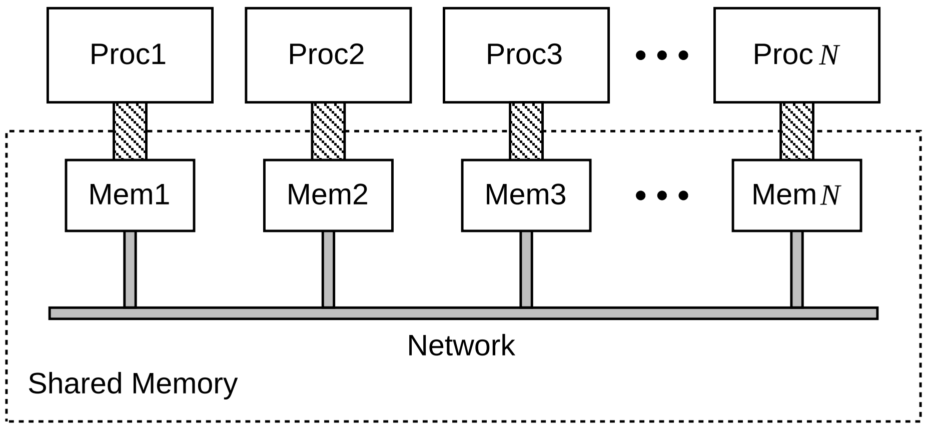

Figure 1 Distributed Shared Memory

describes the implementation of these concepts, and also includes a discussion of the Unix aspects of the implementation. The resulting performance is discussed in Section 4, and compared against earlier work using eager release consistency in Section 5. We discuss related work in Section 6, and conclude in Section 7.

2 Design

TreadMarks' design focuses on reducing the amount of communication necessary to maintain memory consistency. To this end, it presents a *release consistent* memory model [13] to the user. Release consistency requires less communication than conventional, sequentially consistent [15] shared memory, but provides a very similar programming interface. The *lazy* implementation of release consistency in TreadMarks further reduces the number of messages and the amount of data compared to earlier, eager implementations [8]. False sharing is another source of frequent communication in DSM systems. TreadMarks uses *multiple-writer* protocols to address this problem. Multiple-writer protocols require the creation of *diffs*, data structures that record updates to parts of a page. With lazy release consistency, diff creation can often be postponed or avoided, a technique we refer to as *lazy diff creation*.

2.1 Release Consistency

Release consistency (RC) [13] is a *relaxed* memory consistency model that permits a processor to delay making its changes to shared data visible to other processors until certain synchronization accesses occur. Shared memory accesses are categorized either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. Acquires and releases roughly correspond to synchronization operations on a lock, but other synchronization mechanisms can be implemented on top of this model as well. For instance, arrival at a barrier can be modeled as a release, and departure from a barrier as an acquire. Essentially, RC requires ordinary shared memory updates by a processor p to become visible at another processor q, only when a subsequent release by p becomes visible at q.

In contrast, in *sequentially consistent* (SC) memory [15], the conventional model implemented by most snoopy-cache, bus-based multiprocessors, modifications to shared memory must become visible to other processors immediately [15]. Programs written for SC memory produce the same results on an RC memory, provided that (i) all synchronization operations use system-supplied primitives, and (ii) there is a release-acquire pair between conflicting ordinary accesses to the same memory location on different processors [13]. In practice, most shared memory programs require little or no modifications to meet these requirements.

Although execution on an RC memory produces the same results as on a SC memory for the overwhelming majority of the programs, RC can be implemented more efficiently than SC. In the latter, the requirement that shared memory updates become visible immediately implies communication on each write to a shared data item for which other cached copies exist. No such requirement exists under RC. The propagation of the modifications can be postponed until the next synchronization operation takes effect.

2.2 Lazy Release Consistency

In *lazy release consistency* (LRC) [14], the propagation of modifications is postponed *until the time of the acquire*. At this time, the acquiring processor determines which modifications it needs to see according to the definition of RC.

To do so, LRC divides the execution of each process into *intervals*, each denoted by an *interval index*. Every time a process executes a release or an acquire, a new interval begins and the interval index is incremented. Intervals of different processes are partially ordered [1]: (i) intervals on a single processor are totally ordered by program order, and (ii) an interval on processor p precedes an interval on processor q if the interval of q begins with the acquire corresponding to the release that concluded the interval of p. This partial order can be represented concisely by assigning a *vector timestamp* to each interval. A vector timestamp contains an entry for each processor. The entry for processor p in the vector timestamp of interval i of processor p is equal to i. The entry for processor $q \neq p$ denotes the most recent interval of processor q that precedes the current interval of processor p according to the partial order. A processor computes a new vector timestamp at an acquire according to the pair-wise maximum of its previous vector timestamp and the releaser's vector timestamp.

RC requires that before a processor p may continue past an acquire, the updates of all intervals with a smaller vector timestamp than p's current vector timestamp must be visible at p. Therefore, at an acquire, p sends its current vector timestamp to the previous releaser, q. Processor q then piggybacks on the release-acquire message to p, *write notices* for all intervals named in q's current vector timestamp but not in the vector timestamp it received from p.

A write notice is an indication that a page has been modified in a particular interval, but it does *not* contain the actual modifications. The timing of the actual data movement depends on whether an invalidate, an update, or a hybrid protocol is used (see [9]). TreadMarks currently uses an invalidate protocol: the arrival of a write notice for a page causes the processor to invalidate its copy of that page. A subsequent access to that page causes an access miss, at which time the modifications are propagated to the local copy.

Alternative implementations of RC generally cause more communication than LRC. For example, the DASH shared-memory multiprocessor [17] implements RC in hardware, buffering writes to avoid blocking the processor until the write has been performed with respect to main memory and remote caches. A subsequent release is not allowed to perform (i.e., the corresponding lock cannot be granted to another processor) until all outstanding shared writes are acknowledged. While this strategy masks latency, LRC sends far fewer messages, an important consideration in a software implementation on a general-purpose network because of the high per message cost. In an *eager* software implementation of RC [8], a processor propagates its modifications of shared data when it executes a release. This approach also leads to more communication, because it requires a message to be sent to all processors that cache the modified data, while LRC propagates the data only to the next acquirer.

2.3 Multiple-Writer Protocols

False sharing was a serious problem for early DSM systems. It occurs when two or more processors access different variables within a page, with at least one of the accesses being a write. Under the common single-writer protocols, false sharing leads to unnecessary communication. A write to any variable of a page causes the entire page to become invalid on all other processors that cache the page. A subsequent access on any of these processors incurs an access miss and causes the modified copy to be brought in over the network, although the original copy of the page would have sufficed, since the write was to a variable different from the one that was accessed locally. This problem occurs in snoopy-cache multiprocessors as well, but it is more prevalent in software DSM because the consistency protocol operates on pages rather than smaller cache blocks.

To address this problem, Munin introduced a *multiple-writer* protocol [8]. With multiple-writer protocols two or more processors can simultaneously modify their local copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing.

2.4 Lazy Diff Creation

In order to capture the modifications to a shared page, it is initially write-protected. At the first write, a protection violation occurs. The DSM software makes a copy of the page (a *twin*), and removes the write protection so that further writes to the page can occur without any DSM intervention. The twin and the current copy can later be compared to create a *diff*, a runlength encoded record of the modifications to the page.

In TreadMarks, diffs are only created when a processor requests the modifications to a page or a write notice from another processor arrives for that page. In the latter case, it is essential to make a diff in order to distinguish the modifications made by the different processors. This *lazy* diff creation is distinct from Munin's implementation of multiple-writer protocols, where at each release a diff is created for each modified page and propagated to all other copies of the page. The lazy implementation of RC used by TreadMarks allows diff creation to be postponed until the modifications are requested. Lazy diff creation results in a decrease in the number of diffs created (see Section 5) and an attendant improvement in performance.

3 Implementation

3.1 Data Structures

Figure 2 gives an overview of the data structures used. The principal data structures are the *PageArray*, with one entry for each shared page, the *ProcArray*, with one entry for each processor, a set of *interval records* (containing mainly the vector timestamp for that interval), a set of *write notice records*, and a *diff pool*. Each entry in the *PageArray* contains:

1. The current state: no access, read-only access, or read-write access.
2. An *approximate copyset* specifying the set of processors that are believed to currently cache this page.
3. For each page, an array indexed by processor of head and tail pointers to a linked list of *write notice records* corresponding to write notices received from that processor for this page. If the diff corresponding to the write notice has been received, then a pointer to this diff is present in the write notice record. This list is maintained in order of decreasing interval indices.

Each entry in *ProcArray* contains a pointer to the head and the tail of a doubly linked list of *interval records*, representing the intervals of that processor that the local processor knows about. This list is also maintained in order of decreasing interval indices. Each of these interval records contains a pointer to a list of write notice records for that interval, and each write notice record contains a pointer to its interval record.

3.2 Interval and Diff Creation

Logically, a new interval begins at each release and acquire. In practice, interval creation can be postponed until we communicate with another process, avoiding overhead if a lock is reacquired by the same processor. When a lock is released to another processor, or at arrival at a barrier, a new interval is created containing

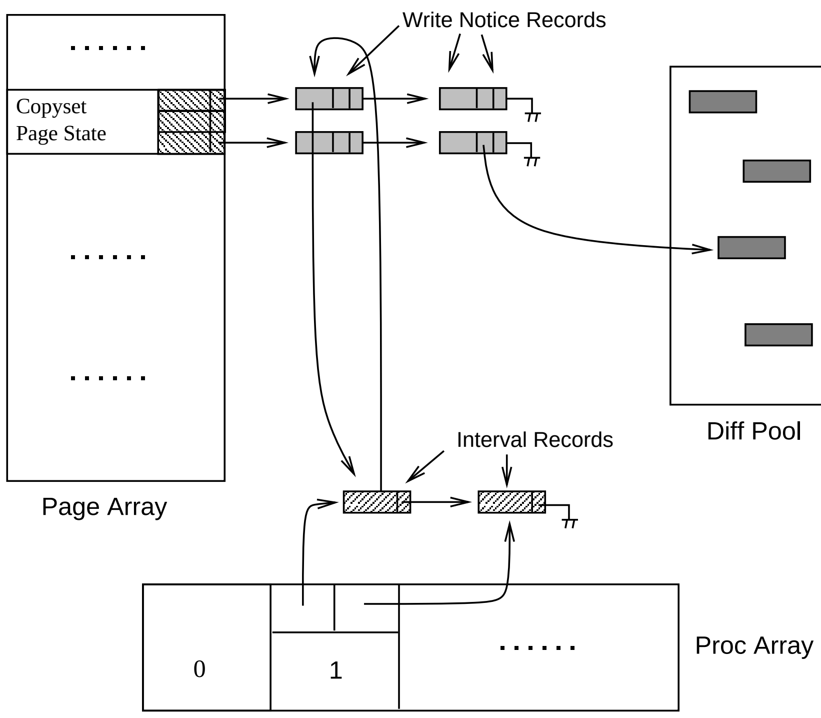

Figure 2 Overview of TreadMarks Data Structures

a write notice for each page that was twinned since the last remote synchronization operation. With lazy diff creation these pages remain writable until a diff request or a write notice arrives for that page. At that time, the actual diff is created, the page is read protected, and the twin is discarded. A subsequent write results in a write notice for the next interval.

3.3 Locks

All locks have a statically assigned manager. Lock management is assigned in a round-robin fashion among the processors. The manager records which processor has most recently requested the lock. All lock acquire requests are directed to the manager, and, if necessary, forwarded to the processor that last requested the lock.

The lock acquire request contains the current vector timestamp of the acquiring processor. The lock request arrives at the processor that either holds the lock or did the last release on it, possibly after forwarding by the lock manager. When the lock is released, the releaser "informs" the acquirer of all intervals between the vector timestamp in the acquirer's lock request message, and the releaser's current vector timestamp. The message contains the following information for each of these intervals:

1. The processor id.
2. The vector timestamp.
3. All write-notices. The write notice in the message is a fixed 16-bit entry containing the page number.

All of this information can easily be derived by following the pointers from the *ProcArray* to the appropriate interval records and from there to the appropriate write notice records.

After receiving this message, the acquirer "incorporates" this information into its data structures. For each interval in the message,

1. the acquirer appends an interval record to the interval record list for that processor, and
2. for each write notice
   (a) it prepends a write notice record to the page's write notice record list, and
   (b) adds pointers from the write notice record to the interval record, and vice versa.

Incorporating this information invalidates the pages for which write notices were received.

3.4 Barriers

Barriers have a centralized manager. At barrier arrival, each client "informs" the barrier manager of its vector timestamp and all of the client's intervals between the last vector timestamp of the manager that the client is aware of (found at the head of the interval record list for the *ProcArray* entry for the manager) and the client's current vector timestamp. When the manager arrives at the barrier, it "incorporates" these intervals into its data structures. When all barrier arrival messages have been received, the manager then "informs" all clients of all intervals between their vector timestamp, as received in their barrier arrival message, and the manager's current vector timestamp. The clients then "incorporate" this information as before. As for locks, incorporating this information invalidates the pages for which write notices were received.

3.5 Access Misses

If the faulting processor does not have a copy of the page, it requests a copy from a member of the page's approximate copyset. The approximate copyset for each page is initialized to contain processor 0.

If write notices are present for the page, the faulting processor obtains the missing diffs and applies them to the page. The missing diffs can be found easily following the linked list of write notices starting from the entry for this page in the `PageArray`. The following optimization minimizes the number of messages necessary to get the diffs. If processor p has modified a page during interval i, then p must have all the diffs of all intervals (including those from processors other than p) that have a smaller vector timestamp than i. It therefore suffices to look at the largest interval of each processor for which we have a write notice but no diff. Of that subset of the processors, a message needs to be sent only to those processors for which the vector timestamp of their most recent interval is not dominated by the vector timestamp of another processor's most recent interval.

After the set of necessary diffs and the set of processors to query have been determined, the faulting processor sends out requests for the diffs in parallel, including the processor id, the page number and the interval index of the requested diffs. When all necessary diffs have been received, they are applied in increasing vector timestamp order.

3.6 Garbage Collection

Garbage collection is necessary to reclaim the space used by write notice records, interval records, and diffs. During garbage collection, each processor validates its copy of every page that it has modified. All other pages, all interval records, all write notice records and all diffs are discarded. In addition, each processor updates the copyset for every page. If, after garbage collection, a processor accesses a page for which it does not have a copy, it requests a copy from a processor in the copyset.

The processors execute a barrier-like protocol, in which processors request and apply all diffs created by other processors for the pages they have modified themselves. Garbage collection is triggered when the amount of free space for consistency information drops below a threshold. An attempt is made to make garbage collection coincide with a barrier, since many of the operations are similar.

3.7 Unix Aspects

TreadMarks relies on Unix and its standard libraries to accomplish remote process creation, interprocessor communication, and memory management. In this section, we briefly describe the implementation of each of these services.

TreadMarks interprocessor communication can be accomplished either through UDP/IP on an Ethernet or an ATM LAN, or through the AAL3/4 protocol on the ATM LAN. AAL3/4 is a connection-oriented, unreliable message protocol specified by the ATM standard. Since neither protocol guarantees reliable delivery, TreadMarks uses operation-specific, user-level protocols on top of UDP/IP and AAL3/4 to insure delivery.

To minimize latency in handling incoming asynchronous requests, TreadMarks uses a `SIGIO` signal handler. Message arrival at any socket used to receive request messages generates a `SIGIO` signal. Since AAL3/4 is a connection-oriented protocol, there is a socket corresponding to each of the other processors. To determine which socket holds the incoming request, the handler for AAL3/4 performs a `select` system call. The handler for UDP/IP avoids the `select` system call by multiplexing all of the other processors over a single receive socket. After the handler receives the message, it performs the request and returns.

To implement the consistency protocol, TreadMarks uses the `mprotect` system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a `SIGSEGV` signal. The `SIGSEGV` signal handler examines the local *PageArray* to determine the page's state. If the local copy is read-only, the handler allocates a page from the pool of free pages and performs a `bcopy` to create a *twin*. Finally, the handler upgrades the access rights to the original page and returns. If the local page is invalid, the handler executes the access miss procedure.

4 Performance

4.1 Experimental Environment

Our experimental environment consists of 8 DECstation-5000/240's running Ultrix V4.3. Each machine has a Fore ATM interface that is connected to a Fore ATM switch. The connection between the interface boards and the switch operates at 100-Mbps; the switch has an aggregate throughput of 1.2-Gbps. The interface board does programmed I/O into transmit and receive FIFOs, and requires fragmentation and reassembly of ATM cells by software. Interrupts are raised at the end of a message or a (nearly) full receive FIFO. All of the machines are also connected by a 10-Mbps Ethernet. Unless otherwise noted, the performance numbers describe 8-processor executions on the ATM LAN using the low-level adaptation layer protocol AAL3/4.

4.2 Basic Operation Costs

The minimum roundtrip time using send and receive for the smallest possible message is 500 μseconds. The minimum time to send the smallest possible message through a socket is 80 μseconds, and the minimum time to receive this message is 80 μseconds. The remaining 180 μseconds are divided between wire time, interrupt processing and resuming the processor that blocked in receive. Using a signal handler to receive the message at both processors, the roundtrip time increases to 670 μseconds.

The minimum time to remotely acquire a free lock is 827 μseconds if the manager was the last processor to hold the lock, and 1149 μseconds otherwise. In both cases, the reply message from the last processor to hold the lock does not contain any write notices (or diffs). The time to acquire a lock increases in proportion to the number of write notices that must be included in the reply message. The minimum time to perform an 8 processor barrier is 2186 μseconds. A remote page fault, to obtain a 4096 byte page from another processor takes 2792 μseconds.

4.3 Applications

We used five programs in this study: Water, Jacobi, TSP, Quicksort, and ILINK. Water, obtained from SPLASH [22], is a molecular dynamics simulation. We made one simple modification to the original program to reduce the number of lock accesses. We simulated 343 molecules for 5 steps. Jacobi implements a form of Successive Over-Relaxation (SOR) with a grid of 2000 by 1000 elements. TSP uses a branch-and-bound algorithm to solve the traveling salesman problem for a 19-city tour. Quicksort sorts an array of 256K integers, using a bubblesort to sort subarrays of less than 1K elements. ILINK, from the LINKAGE package [16], performs genetic linkage analysis (see [10] for more details). ILINK's input consists of data on 12 families with autosomal dominant nonsyndromic cleft lip and palate (CLP).

4.4 Results

Figure 3 presents speedups for the five applications. The speedups were calculated using uniprocessor times obtained by running the applications without TreadMarks. Figure 4 provides execution statistics for each of the five applications when using 8 processors.

The speedup for Water is limited by the high communication (798 Kbytes/second and 2238 messages/second) and synchronization rate (582 lock accesses/second). There are many short messages (the average message size is 356 bytes), resulting in a large communication overhead. Each molecule is protected by a lock that is accessed frequently by a majority of the processors. In addition, the program uses barriers for synchronization.

Jacobi exclusively uses barriers for synchronization. Jacobi's computation to communication ratio is an order of magnitude larger than that of Water. In addition, most communication occurs at the barriers and between neighbors. On the ATM network, this communication can occur in parallel. The above two effects compound, resulting in near-linear speedup for Jacobi.

TSP is an application that exclusively uses locks for synchronization. Like Jacobi, TSP has a very high computation to communication ratio, resulting in near-linear speedup. While the number of messages per second is slightly larger than for Jacobi, TSP transmits only a quarter of the amount of data transmitted by Jacobi.

Quicksort also uses locks for synchronization. Quicksort's synchronization rate is close to that of Jacobi's. It, however, sends over twice as many messages and data per second, resulting in slightly lower, although good, speedups. The number of kilobytes per second transmitted by Quicksort is similar to that transmitted by Water, but it sends 3 times fewer messages and the number of synchronization operations is an order of magnitude lower than for Water. As a result, speedup for Quicksort is higher than for Water.

ILINK achieves less than linear speedup on TreadMarks because of a load balancing problem inherent to the nature of the algorithm [10]. It is not possible to predict in advance whether the set of iterations distributed to the processors will result in the same amount of work on each processor, without significant

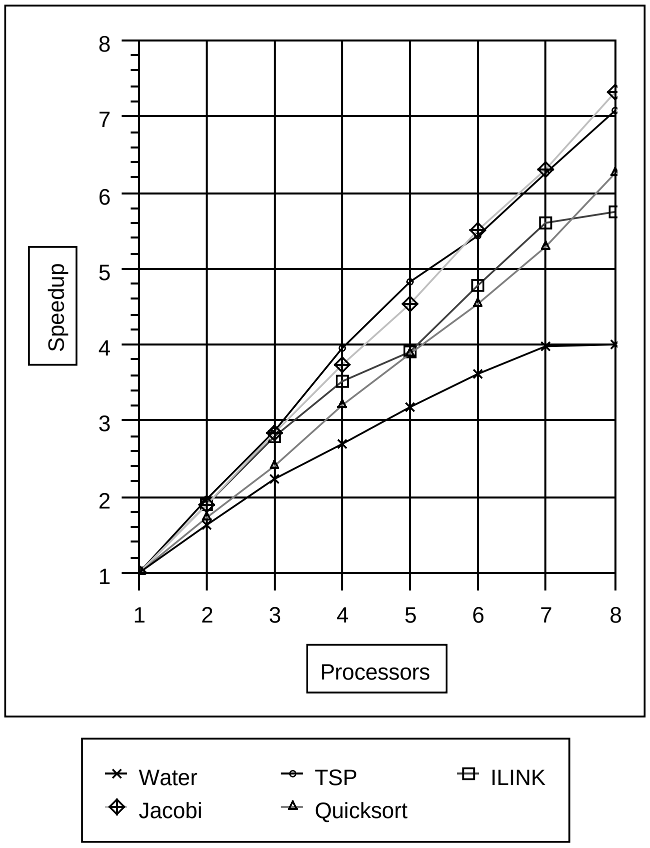

Figure 3 Speedups Obtained on TreadMarks

	Water	Jacobi	TSP	Quicksort	ILINK
Input	343 mols 5 steps	2000x1000 floats	19-city tour	256000 integers	CLP
Time (secs)	15.0	32.0	43.8	13.1	1113
Barriers/sec	2.5	6.3	0	0.4	0.4
Locks/sec	582.4	0	16.1	53.9	0
Msgs/sec	2238	334	404	703	456
Kbytes/sec	798	415	121	788	164

Figure 4 Execution Statistics for an 8-Processor Run on TreadMarks

computation and communication. Consequently, speedups are somewhat lower than one would expect based on the communication and synchronization rates.

4.5 Execution Time Breakdown

Figure 5 shows a percentage breakdown of the execution times for 8-processor versions of all 5 applications. The “Computation” category is the time spent executing application code; “Unix” is the time spent executing Unix kernel and library code; and “TreadMarks” is the time spent executing TreadMarks code. “Idle Time” refers to the time that the processor is idle. Idle time results from waiting for locks and barriers, as well as from remote communication latency.

The largest overhead components are the Unix and idle times. The idle time reflects to some extent the amount of time spent waiting for Unix and TreadMarks operations on other nodes. The TreadMarks overhead is much smaller than the Unix overhead. The largest percentage TreadMarks overhead is for Water (2.9% of overall execution time). The Unix overhead is at least three times as large as the TreadMarks overhead for all the applications, and is 9 times larger for ILINK.

Figure 6 shows a breakdown of the Unix overhead. We divide Unix overhead into two categories: communication and memory management. Communication overhead is the time spent executing *kernel* operations to support communication. Memory management overhead is the time spent executing *kernel* operations to support the *user-level* memory management, primarily page protection changes. In all cases, at least 80% of the kernel execution time is spent in the communication routines, suggesting that cheap communication is the primary service a software DSM needs from the operating system.

Figure 7 shows a breakdown of TreadMarks overhead. We have divided the overhead into three categories: memory management, consistency, and “other”. “Memory management” overhead is the time spent at the *user-level* detecting and capturing changes to shared pages. This includes twin and diff creation and diff

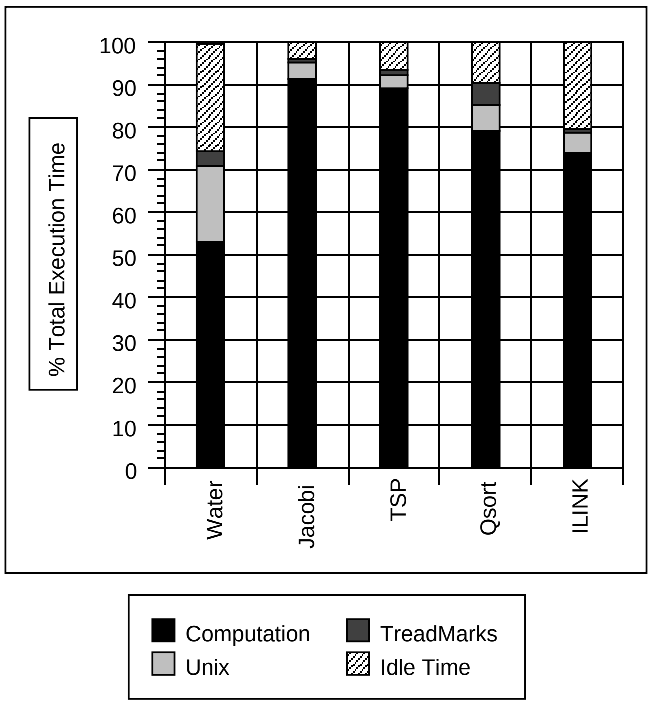

Figure 5 TreadMarks Execution Time Breakdown

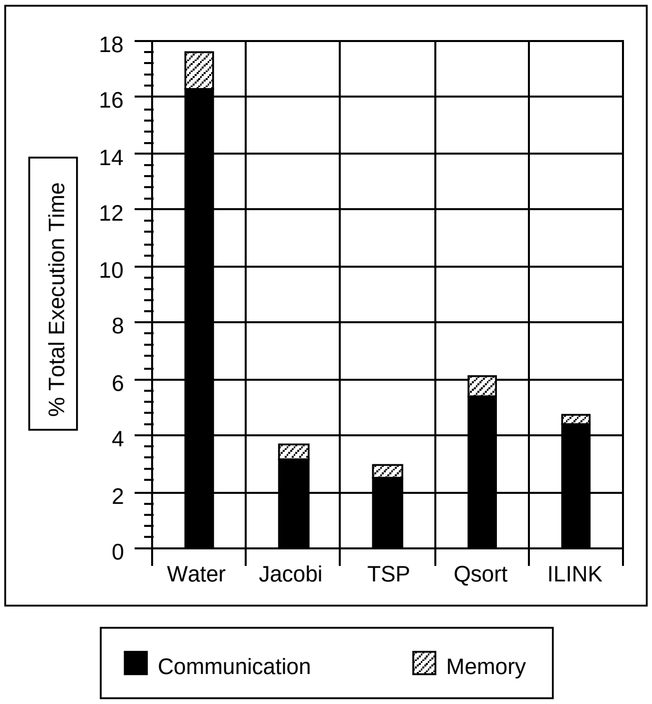

Figure 6 Unix Overhead Breakdown

application. "Consistency" is the time spent propagating and handling consistency information. "Other" consists primarily of time spend handling communication and synchronization. TreadMarks overhead is dominated by the memory management operations. Maintaining the rather complex partial ordering between intervals adds only a small amount to the execution time.

4.6 Effect of Network and Communication Protocol

We ran Water, the application the highest communication overhead on two other communication substrates: UDP over the ATM network, and UDP over an Ethernet. Figure 8 shows the total 8-processor execution times for all three different communication substrates and a breakdown into computation, Unix overhead, TreadMarks overhead, and idle time.

Overall execution time increases from 15.0 seconds on ATM-AAL3/4 to 17.5 seconds on ATM-UDP and to 27.5 seconds on Ethernet-UDP. Computation time and TreadMarks overhead remain constant, Unix overhead increases slightly, but the idle time increases from 3.9 seconds on AAL3/4 to 5.0 seconds on ATM/UDP, and to 14.4 seconds over the Ethernet. The increase from ATM-AAL3/4 to ATM-UDP is due to increased protocol overhead in processing network packets. For the Ethernet, however, it is largely due to network saturation.

4.7 Summary

TreadMarks achieves good speedups for Jacobi, TSP, Quicksort, and ILINK on the 100 Mbit/sec ATM LAN. For a slightly modified version of the Water program from the Splash benchmark suite, TreadMarks achieved only a moderate speedup, because of the large number of small messages.

The overhead of the DSM is dominated by the communication primitives. Since wire time is negligible on the ATM LAN for our applications, the greatest potential to improve overall performance is reducing

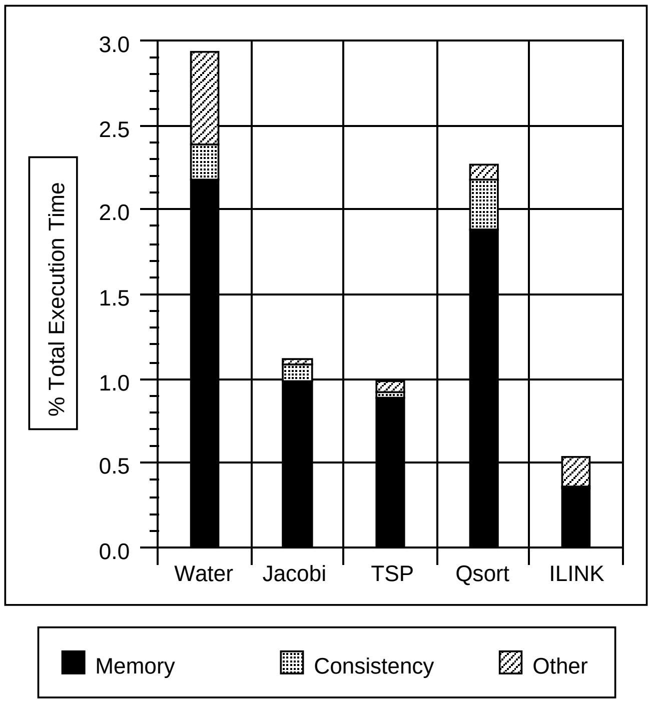

Figure 7 TreadMarks Overhead Breakdown

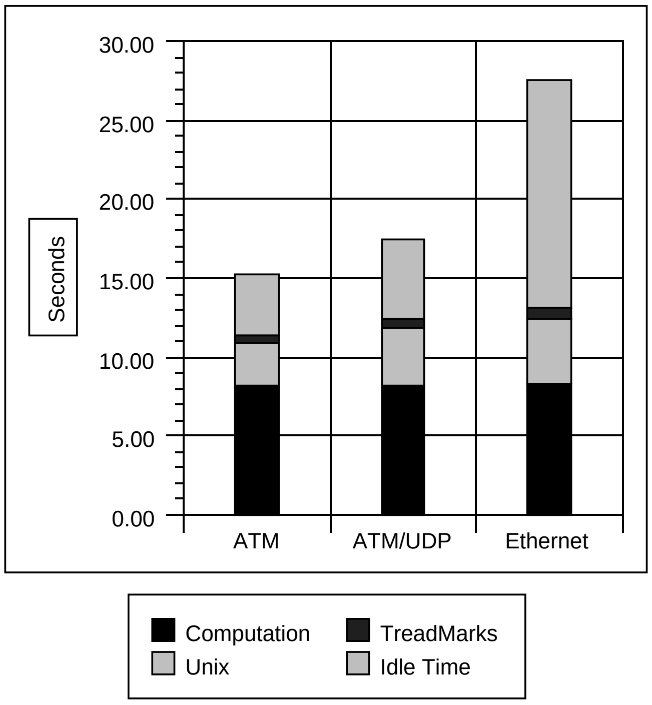

Figure 8 Execution Time for Water

the *software* communication overhead. Although the use of the lightweight AAL3/4 protocol reduces the total send and receive time, these are only a part of the overall communication overhead. Lower-overhead user-level communications interfaces or a kernel-level implementation would improve performance.

A kernel implementation of the memory management would have little effect on overall performance. In the worst case (Water), TreadMarks spent less than 2.2% of its time detecting and capturing changes to shared pages. Most of this time is spent copying the page and constructing the diff. Less than 0.8% of the time is spent in the kernel generating the signal or performing the `mprotect`.

5 Lazy vs. Eager Release Consistency

5.1 Eager Release Consistency: Design and Implementation

We implemented an *eager* version of RC (ERC) to assess the performance differences between ERC and LRC. At the time of a release, ERC creates diffs of modified pages, and distributes each diff to all processors that cache the corresponding page. Our implementation of ERC uses an update protocol. Eager invalidate protocols have been shown to result in inferior performance for DSM systems [14]. We are thus comparing LRC against the best protocol available for ERC. With an eager invalidate protocol, the diffs cause a large number of invalidations, which trigger a large number of access misses. In order to satisfy these access misses, a copy of the entire page must be sent over the network. In contrast, lazy invalidate protocols only move the diffs, because they maintain enough consistency information to reconstruct valid pages from the local (out-of-date) copy and the diffs.

5.2 Performance

Figures 9 to 12 compare the speedups, the message and data rates, and the rate of diff creation between the eager and lazy version of the five applications. In order to arrive at a fair comparison of the message and the data rate, we normalize these quantities by the average execution time of ERC and LRC.

LRC performs better than ERC for Water and Quicksort, because the LRC sends fewer messages and a smaller amount of data. In Water, in particular, ERC sends a large number of updates at each release, because all processors have copies of most of the shared data.

Jacobi performs slightly better under LRC than under ERC. Although communication requirements are similar in both cases, Figure 12 shows that the lazy diff creation of LRC generates 25% fewer diffs than ERC, thereby decreasing the overhead. For ILINK, performance is comparable under both schemes.

For TSP, ERC results in better performance than LRC. TSP is implemented using a branch-and-bound algorithm that uses a *current minimum* to prune searching. The performance on LRC suffers from the fact that TSP is not a *properly labeled* [13] program. Although updates to the current minimum tour length are synchronized, read accesses are not. Since LRC updates cached values only on an *acquire*, a processor may read an old value of the current minimum. The execution remains correct, but the work performed by the processor may be redundant since a better tour has already been found elsewhere. With ERC, this is less likely to occur since ERC updates cached copies of the minimum when the lock protecting the minimum is released. By propagating the bound earlier, ERC reduces the amount of redundant work performed, leading to a better speedup. Adding synchronization around the read accesses would deteriorate performance, given the very large number of such accesses.

6 Related Work

Among the many proposed relaxed memory consistency models, we have chosen release consistency [13], because it requires little or no change to existing shared memory programs. An interesting alternative is *entry consistency* (EC) [4]. EC differs from RC in that it requires all shared data to be explicitly associated with some synchronization variable. On a lock acquisition EC only propagates the shared data associated with that lock. EC, however, requires the programmer to insert additional synchronization in shared memory programs to execute correctly on an EC memory. Typically, RC does not require additional synchronization.

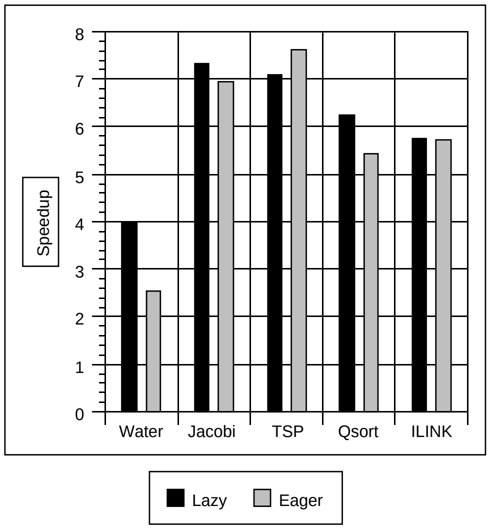

Figure 9 Comparison of Lazy and Eager Speedups

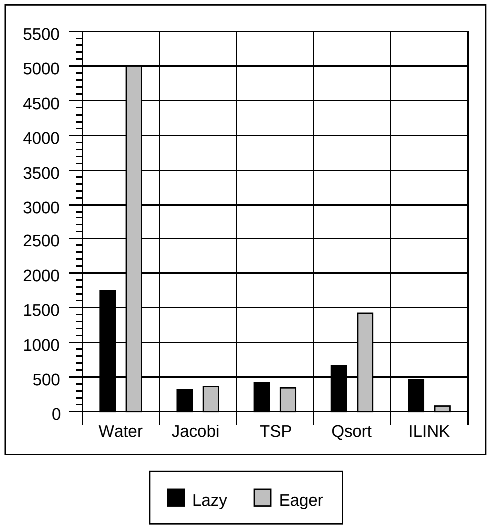

Figure 10 Message Rate (messages/sec)

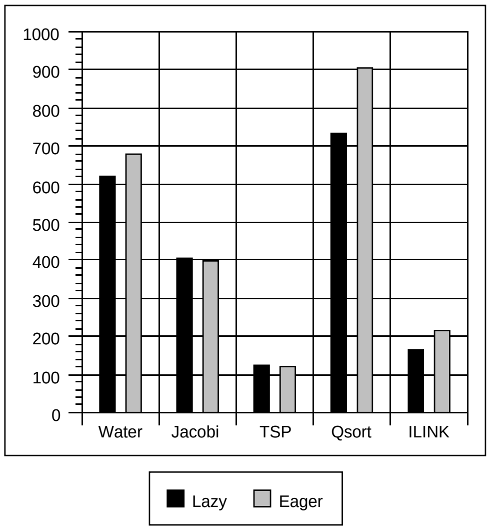

Figure 11 Data Rate (kbytes/sec)

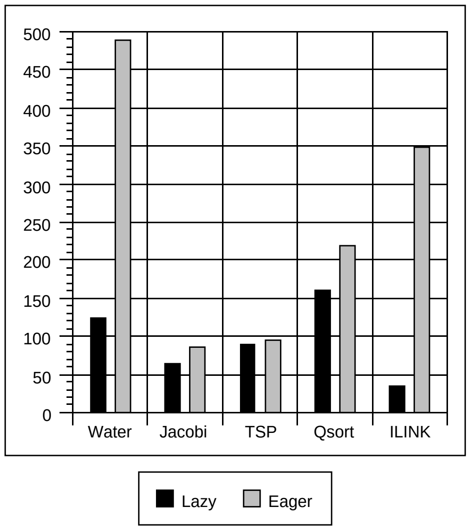

Figure 12 Diff Creation Rate (diffs/sec)

In terms of comparisons with other systems, we restrict ourselves to implementations on comparable processor and networking technology. Differences in processor and network speed and their ratio lead to different tradeoffs [9], and makes comparisons with older systems [3, 8, 11, 12, 18, 21] difficult. We have however borrowed from Munin [8] the concept of multiple-writer protocols. Munin also implements eager release consistency, which moves more messages and data than lazy release consistency.

Bryant et al. [7] implemented SSVM (Structured Shared Virtual Memory) on a star network of IBM RS-6000s running Mach 2.5. Two different implementation strategies were followed: one using the Mach external pager interface [24], and one using the Mach exception interface [5]. They report that the latter implementation—which is very similar to ours—is more efficient, because of the inability of Mach's external pager interface to asynchronously update a page in the user's address space. Also, the time to update a page in a user's address space is higher for the external pager interface than for the exception interface (1.2 vs. 0.7 milliseconds) because the need for a `data_request - data_provided` message transaction when using the external pager interface. The overhead of a page fault (without the actual page transfer) is approximately 1 milliseconds, half of which is attributed to process switching overhead in the exception-based implementation. The time to transfer a page (11 milliseconds) dominates all other overheads in the remote page fault time.

Bershad et al. [4] use a different strategy to implement EC in the Midway DSM system, running on DECStation-500/200s connected by an ATM LAN and running Mach 3.0. Instead of relying on the VM system to detect shared memory updates, they modify the compiler to update a software dirty bit. Our results show that, at least in Ultrix and we suspect in Mach as well, the software communication overhead dominates the memory management overhead.

DSVM6K [6] is a sequentially consistent DSM system running on IBM RS/6000s connected by 220-Mbps fiber optic links and a nonblocking crossbar switch. The system is implemented inside the AIX v3 kernel and uses a low-overhead protocol for communication over the fiber optic links (IMCS). A remote page fault takes 1.75 milliseconds when using IMCS, and is estimated to take 3.25 milliseconds when using TCP/IP. The breakdown of the 1.75 milliseconds page fault time is: 1.05 milliseconds for DSVM6K overhead, 0.47 milliseconds for IMCS overhead and 0.23 milliseconds of wire time. Shiva [19] is an implementation of sequentially consistent DSM on an Intel IPSC/2. Shiva is implemented outside the kernel. A remote page fault takes 3.82 milliseconds, and the authors estimate that time could be reduced by 23 percent by a kernel implementation. In comparison, our page fault times are 2.8 milliseconds using AAL3/4. While these numbers are hard to compare because of differences in processor and networking hardware, our results highlight the cost of the software communication overhead. Either an in-kernel implementation or fast out-of-kernel communication interfaces need to be provided in order to build an efficient DSM system.

7 Conclusions

Good performance has been achieved for DSM systems built on various research operating systems. However, in order to use DSM as a platform for parallel computation on clusters of workstations, efficient user-level implementations must be available on commercial operating systems. It is with this goal in mind that we set out to conduct the experiment described in this paper.

We implemented a DSM system at the user level on DECstation-5000/240's connected to a 100-Mbps ATM LAN and a 10-Mbps Ethernet. We focused our implementation efforts on reducing the cost of communication, using techniques such as lazy release consistency, multiple-writer protocols, and lazy diff creation. On the ATM network, we avoided the overhead of UDP/IP by using the low-level AAL3/4 protocol.

On the ATM network, we achieved good speedups for Jacobi, TSP, Quicksort, and ILINK, and moderate speedups for a slightly modified version of Water. Latency and bandwidth limitations reduced the speedups by varying amounts on the Ethernet. We conclude that user-level DSM is a viable technique for parallel computation on clusters of workstations connected by suitable networking technology.

In order to achieve better DSM performance for more fine-grained programs like Water, the software communication overhead needs to be reduced through lower-overhead communication interfaces and implementations.

References

[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] S. Ahuja, N. Carreiro, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. A distributed implementation of the shared data-object model. *Distributed Systems and Multiprocessor Workshop*, pages 1–19, 1989.

[4] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the '93 CompCon Conference*, pages 528–537, February 1993.

[5] D. Black, D. Golub, R. Rashid, A. Tevanian, and M. Young. The Mach exception handling facility. *SigPlan Notices*, 24(1):45–56, May 1988.

[6] M.L. Blount and M. Butrico. DSVM6K: Distributed shared virtual memory on the Risc System/6000. In *Proceedings of the '93 CompCon Conference*, pages 491–500, February 1993.

[7] R. Bryant, P. Carini, H.-Y. Chang, and B. Rosenburg. Supporting structured shared virtual memory under Mach. In *Proceedings of the 2nd Mach Usenix Symposium*, November 1991.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] 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 International Symposium on Computer Architecture*, pages 244–255, May 1993.

[10] S. Dwarkadas, A. A. Schäffer, R. W. Cottingham Jr., A. L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. To appear in Journal of Human Heredity, 1993.

[11] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[12] A. Forin, J. Barrera, and R. Sanzi. The shared memory server. In *Proceedings of the 1989 Winter Usenix Conference*, pages 229–243, December 1989.

[13] 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 International Symposium on Computer Architecture*, pages 15–26, May 1990.

[14] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

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

[16] G.M. Lathrop, J.M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proceedings of National Academy of Science*, 81:3443–3446, June 1984.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

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

[19] K. Li and R. Schaefer. A hypercube shared virtual memory system. *1989 International Conference on Parallel Processing*, 1:125–131, 1989.

[20] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[21] U. Ramachandran and M.Y.A. Khalidi. An implementation of distributed shared memory. *Software: Practice and Experience*, 21(5):443–464, May 1991.

[22] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[23] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency:Practice and Experience*, 2(4):315–339, December 1990.

[24] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, October 1987.