An Evaluation of Software-Based Release Consistent Protocols

Pete Keleher

Department of Computer Science
The University of Maryland
College Park, MD 20742

Alan L. Cox, Sandhya Dwarkadas, Willy Zwaenepoel

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

Abstract

This paper presents an evaluation of three software implementations of release consistency. Release consistent protocols allow data communication to be aggregated, and multiple writers to simultaneously modify a single page. We evaluated an eager invalidate protocol that enforces consistency when synchronization variables are released, a lazy invalidate protocol that enforces consistency when synchronization variables are acquired, and a lazy hybrid protocol that selectively uses update to reduce access misses.

Our evaluation is based on implementations running on DECstation-5000/240s connected by an ATM LAN, and an execution-driven simulator that allows us to vary network parameters. Our results show that the lazy protocols consistently outperform the eager protocol for all but one application, and that the lazy hybrid performs the best overall. However, the relative performance of the implementations is highly dependent on the relative speeds of the network, processor, and communication software. Lower bandwidths and high per byte software communication costs favor the lazy invalidate protocol, while high bandwidths and low per byte costs favor the hybrid. Performance of the eager protocol approaches that of the lazy protocols only when communication becomes essentially free.

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.

1 Introduction

Software distributed shared memory (DSM) [13] enables processes on different machines to share memory, even though the machines physically do not share memory. DSM is an appealing approach for parallel programming on networks of workstations, because most programmers find it easier to use than message passing, which requires them to explicitly partition data and manage communication.

Early DSM systems suffered from performance problems because they required large amounts of communication. These early designs implemented the shared memory abstraction by imitating consistency protocols used by bus-based hardware shared memory multiprocessors. The low latencies on these bus-based machines allowed them to implement *sequential consistency* (SC) [11], but with the much higher latencies present on networks sequential consistency causes serious inefficiencies. Furthermore, given the large consistency units in DSM (virtual memory pages), false sharing was a serious problem for many applications.

In order to address the performance problems with earlier DSM systems, relaxed memory models, such as *release consistency* (RC) [8], were introduced into DSM systems [5]. With very little change to the programming model, RC permits several runtime optimizations that reduce the amount of communication. In particular, it allows the protocol to aggregate the transmission of shared memory writes until a later synchronization point. Furthermore, it permits the use of *multiple-writer protocols* [5], allowing multiple, simultaneous writes by different processors to the same page, thereby reducing the impact of false sharing.

This paper evaluates three different software implementations of RC on a network of workstations: an *eager invalidate*(EI) protocol, a *lazy invalidate*(LI) protocol, and a *lazy hybrid*(LH) protocol. Eager protocols enforce RC when a synchronization variable is released. Lazy protocols enforce RC when a synchronization variable is acquired. Both EI and LI invalidate remote copies of modified data, while LH uses a combination of invalidate and update. We do not consider eager update or pure lazy update protocols, because earlier work [6] has shown that eager update performs comparably to EI, and lazy update performs substantially worse than LI or LH.

We explore the trade-offs between these three protocols by measurement and simulation. EI involves less computational overhead, but for most applications it sends more messages and data than LI and LH. Comparing the two lazy protocols, LI is more efficient in terms of computation and the amount of data moved, but it sends more messages than LH.

Our measurement results were obtained using TreadMarks, an efficient user-level DSM system for standard Unix systems. By default, TreadMarks uses LI, but we modified the implementation to also include the other two protocols. Our hardware is a network of 8 DECstation-5000/240's that are connected by a 100-Mbps switch-based ATM LAN. Overall, the results show that a software DSM has good performance for a variety of programs. LH achieves speedups of 7.5 for SOR, 7.2 for TSP, 5.8 for ILINK, 5.7 for IS, 5.7 for MIP, 4.5 for Water, 3.6 for FFT, and 3.3 for Barnes-Hut. The performance of LI is comparable. EI generally performs worse. The differences are largest for Barnes-Hut, IS, TSP, and Water. EI performs better for FFT than either LI or LH.

We then vary the communication overhead and the bandwidth using a parallel, execution-driven simulator. In order to accurately compare the different protocols, our simulations include the execution of the actual TreadMarks code. This simulator was validated against the implementation and was found to be accurate to within 10%. With the exception of FFT, the lazy protocols consistently outperform EI. At low bandwidth, regardless of the per byte software communication cost, the LI protocol outperforms the others because it sends less data. LH performs better in cases of relatively static sharing behavior where runtime predictions of access patterns are possible, especially at a high bandwidth or a low per byte software communication cost. In this case, the elimination of access miss messages outweighs the cost of transferring some unused data, the cost of which is decreased at the higher bandwidths.

The outline of the rest of this paper is as follows. Section 2 elaborates on the definition of RC and the three protocols, EI, LI, and LH. Section 3 summarizes some of the implementation aspects. The resulting performance is discussed in Section 4. In Section 5, we present a simulation-based analysis of the trade-offs among the protocols as the ratio of network to processor speed, as well as the cost of communication are varied. We discuss related work in Section 6, and conclude in Section 7.

2 Release Consistency Protocols

Release consistency requires less communication than the canonical memory model, sequential consistency [11], but provides a very similar programming interface. An *eager* implementation [5] of release consistency enforces consistency when a synchronization variable is released. In contrast, *lazy* implementations of release consistency enforce consistency when synchronization variables are acquired. Strictly speaking, lazy protocols implement a slight weaker memory model than EI. However, the difference is irrelevant for all of the programs in this study except TSP, where EI's memory model is slightly favored. False sharing is another source of frequent communication in DSM systems. The use of *multiple-writer* protocols addresses this problem. Multiple-writer protocols require the creation of *diffs*, data structures that record updates to parts of a page.

2.1 Release Consistency

RC 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 *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. In order for an RC protocol to guarantee correctness, all synchronization must go through system-visible synchronization operations. 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 is represented 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, no later than the time when a subsequent release by p becomes visible at q.

In contrast, in SC memory, the conventional model implemented by most snoopy-cache, bus-based multiprocessors, modifications to shared memory must become visible to other processors immediately. 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 [8]. 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 an 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 Multiple-Writer Protocols

To address the problem of false sharing — concurrent accesses to unrelated items in the same page — all of the protocols described in this paper are *multiple-writer* protocols. In a multiple-writer protocol two or more processors can simultaneously modify their local copies of the same shared page. The concurrent modifications are merged at synchronization points, in accordance with the definition of RC.

Modifications are summarized as *diffs*. Figure 1 shows how diffs are created and applied. Shared pages are initially write-protected, causing a protection violation to occur when a page is first written. 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 DSM intervention. Differences between the twin and a later copy of the page can then be used to create a *diff*,

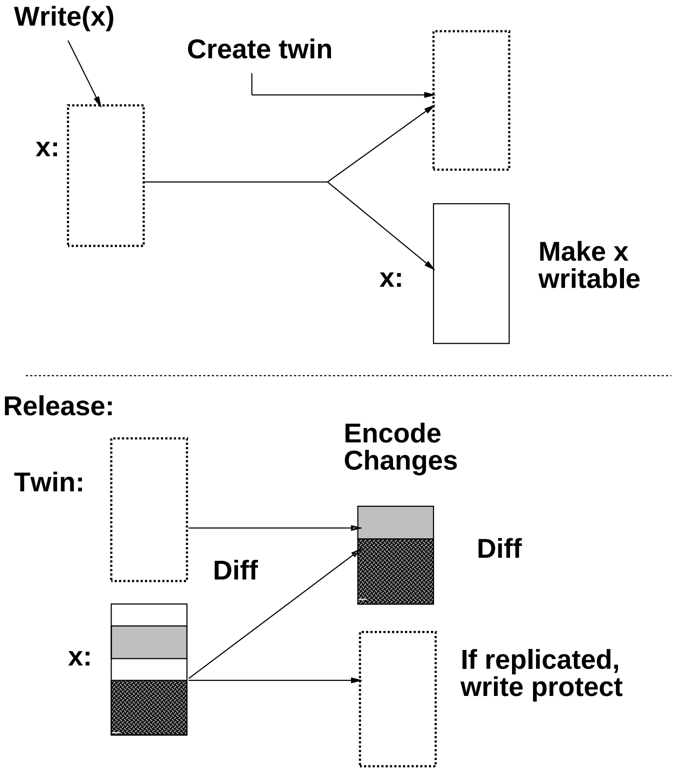

Figure 1 Diff Creation

a runlength encoded record of the modifications made to the page.

2.3 The Eager Invalidate Protocol

In an eager protocol, modifications to shared data are made visible globally at the time of a release. The EI protocol, in particular, attempts to invalidate remote copies of any page that has been modified locally. If the remote copy of the page is READ_ONLY, then it is simply invalidated. If it is in a READ_WRITE state, the remote site appends a diff describing its modifications to the reply message and then invalidates the page. Diffs received in the replies from invalidates are applied to the local copy. If an invalidate is received for a page that is currently being flushed, each of the processors performing the flush creates a diff describing its local modifications. These diffs, together with any diffs received from replies (from processors that are not concurrently flushing the page), are then sent to all other processors in the system. More efficient solutions could be designed for the case of concurrent flushes, but this situation arises rarely.

The EI protocol uses an approximate *copyset* to determine the remote locations to be invalidated. A copyset is a bitmask indicating which processors have a copy of the page. Since this local copyset may not be up to date, the acknowledgement to an invalidate message also contains the remote site's version of the page's copyset. If the local site thereby learns of other processors caching a modified page, additional protocol rounds are used to ensure that all remote copies are invalidated. In practice, flushes rarely take more than a single round.

On an access miss, the faulting processor fetches the entire page from the processor that last modified the page.

2.4 The Lazy Invalidate Protocol

With a lazy protocol, 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. The execution of each process is divided 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 on q begins with the acquire corresponding to the release that concluded the interval on 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 from q, the updates of all intervals with a smaller vector timestamp than q'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. In LI, arrival of a write notice causes the corresponding page to be invalidated.

Diffs are created when a processor requests the modifications to a page, or a write notice from another processor arrives for a dirty page. In the latter case, it is essential to make a diff in order to distinguish the modifications made by the different processors.

Access to an invalidated page causes an access miss. At this point, the faulting processor must retrieve and apply to the page all diffs that were created during intervals that precede the faulting interval in the partial order. 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 requests the diffs in parallel. When all necessary diffs have been received, they are applied in increasing vector timestamp order.

2.5 The Lazy Hybrid Protocol

LH is a lazy protocol similar to LI, but instead of invalidating the modified pages, it updates some of the pages at the time of an acquire. LH attempts to exploit temporal locality by assuming that any page accessed by a processor in the past will probably be accessed by that processor again in the future. All pages that are known to have been accessed by the acquiring processor are therefore updated. Thus, for applications with fairly static sharing patterns, the communication required can be optimized with the help of this protocol.

Each processor uses a copyset to track accesses to pages by other processors. The copyset is used to determine whether a given diff must be sent to a remote location. However, we also need to determine the set of diffs to be examined. There are several possibilities for determining this set, from only those diffs created during the previous interval by the releasing processor to some notion of every possible diff. We investigated several variations, but found that the heuristic that works best is to look at every diff pertaining to a write notice that is sent. For each such write notice, if the releasing processor has the diff and the acquiring processor is in the local copyset for that page, the diff is appended to the lock grant message.

Diffs are created as in LI, but diffs also may need to be created when it is decided that they need to be appended on a lock grant message.

On arrival at a barrier, each processor creates a list describing local write notices that may not have been seen by other processors. A list for processor p_j at processor p_i consists of processor p_i's notion of all local write notices that have not been seen by p_j. p_i sends an update message(s) to p_j containing all the diffs corresponding to write notices in this list. Unlike eager flushes, the barrier updates do not have to be acknowledged because lost updates will simply result in access misses.

2.6 Protocol Trade-Offs

LI and LH generally require fewer messages than EI, especially for programs that use locks. The primary advantage of the lazy protocols during lock transfers is that communication is limited to the two synchronizing processes. A release in an eager system often requires invalidations to be sent to processes otherwise uninvolved in the synchronization. EI's invalidations can also result in a larger number of remote access misses due to false sharing. Since LI and LH usually exchange data in the form of diffs, the total amount of data exchanged is usually less than for EI, because EI moves entire pages in the common case. Comparing LI and LH, LI experiences more access misses, and therefore sends more messages. LI, however, sends less data, because LH may send unnecessary data at the time of an acquire.

EI's invalidations may also increase lock acquisition latency because releases cannot take place until the invalidations have been sent and acknowledged. Lock transfer in LI and LH, in contrast, only involves communication between the releasing and the acquiring processor. LH often appends updates to lock grant messages, and the extra time required to generate and process this data can slow down the lock acquisition.

EI is substantially less complex than LI and LH. As soon as a release has been completed, all state concerning the modified page in EI (twin, diff, etc.) can be discarded. There is also no need to move information transitively, as all information is immediately made globally visible. Finally, EI creates far fewer diffs than LI, which in turn creates fewer diffs than LH.

The choice between these three protocols thus involves a complex trade-off between the number of access misses, the number of messages, the amount of data, the lock acquisition times, and the protocol overhead. Table 1 summarizes these trade-offs.

	Lock Latency	Remote Access Misses	Msgs	Data	Diffs	Protocol Complexity
Eager-Inv (EI)	Low	High	High	High	Low	Low
Lazy-Inv (LI)	Low	Medium	Medium	Low	Medium	Medium
Lazy-Hyb (LH)	Medium	Low	Low	Medium	High	Medium

Table 1 Protocol Trade-offs

3 Implementation

3.1 TreadMarks

The three protocols described in Section 2 were implemented in the TreadMarks DSM system. TreadMarks programs follow a conventional shared memory style, using threads to express parallelism and locks and barriers to synchronize. TreadMarks is entirely implemented as a C library, using an interface similar to the `parmacs` macros from Argonne National Laboratory [14] for thread and synchronization support.

To provide for a fair comparison, the three protocols share as much code as possible. In particular, the same primitives are used for communication (sockets and SIGIO signals) and for memory management (`mprotect` and SIGSEGV signals). The diff creation mechanism, and the lock and barrier implementations are identical.

None of the protocols is overly complex to implement. The entire system takes about 4000 lines of code. Approximately 1200 lines are specific to the lazy protocols, and an additional 300 lines are specific to LH. EI is fully implemented in only 800 lines of code. For a more detailed discussion of the implementation of the three protocols, we refer the reader to Keleher's Ph.D. dissertation [9].

3.2 Experimental Environment

Our experimental environment consists of 8 DECstation-5000/240's running Ultrix V4.3. Each machine has a Fore ATM interface 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 messages to be assembled and disassembled from 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.

3.3 Basic Operation Costs

The minimum round-trip time using send and receive for the smallest possible message is 500 μseconds. Sending a minimal message takes 80 μseconds, receiving it takes a further 80 μseconds, and the remaining 180 μseconds are divided between wire time, interrupt processing and resuming the processor that blocked on a receive. Using a signal handler to receive the message at both processors, the round-trip time increases to 670 μseconds.

The minimum time to remotely acquire a free lock is 827 μseconds if the lock 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 access miss, to obtain a 4096 byte page from another processor, takes 2792 μseconds.

3.4 Applications

The eight programs used in this study vary considerably in size and complexity. `SOR` (Successive Over-Relaxation) and `TSP` (Traveling Salesman Problem) are small programs, developed locally. `Water` and `Barnes-Hut` come from the Stanford Parallel Applications for Shared Memory (SPLASH) benchmark suite [15]. `FFT` (Fast Fourier Transform) and `IS` (Integer Sort) are taken from the NAS benchmark suite [2]. Finally, `ILINK` (genetic linkage) [7] and `MIP` (mixed integer programming) are large programs, each more than ten thousand lines of code. Parallel versions of both programs were developed locally.

Table 2 summarizes the applications and their input sets. `Syncs_per_second` is the synchronization rate for an eight processor run under LI.

4 Performance Measurements

We first compare the speedups of the programs in our application suite for EI, LI, and LH. We then present a breakdown of the execution times into component costs in order to distinguish between costs due to the protocol and those due to the underlying operating system and hardware.

Program	Input	Sync. Type	Syncs. Per Second
SOR	2000 x 1000 floats	barriers	51
TSP	19 cities	locks	16
Water	343 molecules	locks, barriers	661
Barnes	4096 bodies	barriers	2
FFT	64 x 64 x 32	barriers	13
IS	$N = 2^{20}, B_{max} = 2^7$	locks, barriers	228
ILINK	CLP	locks	3
MIP	misc05.mps	locks	531

Table 2 Application Suite

4.1 Speedup Comparison

Figure 2 presents the speedups on 8 processors for the eight applications for each of the three protocols. In all cases, speedup was calculated with reference to the same code run single-threaded with the TreadMarks library calls removed. Table 3 shows rate statistics for the three protocols. We use rate statistics rather than totals in order to make meaningful comparisons between applications that vary widely in running times. `Total_Msgs` is the overall rate at which messages are sent. `Data` is the amount of data sent per second, in kilobytes. `Access_Misses` is the number of access misses per second that required remote communication. Finally, `Diffs_Created` is the rate at which diffs were created in the system.

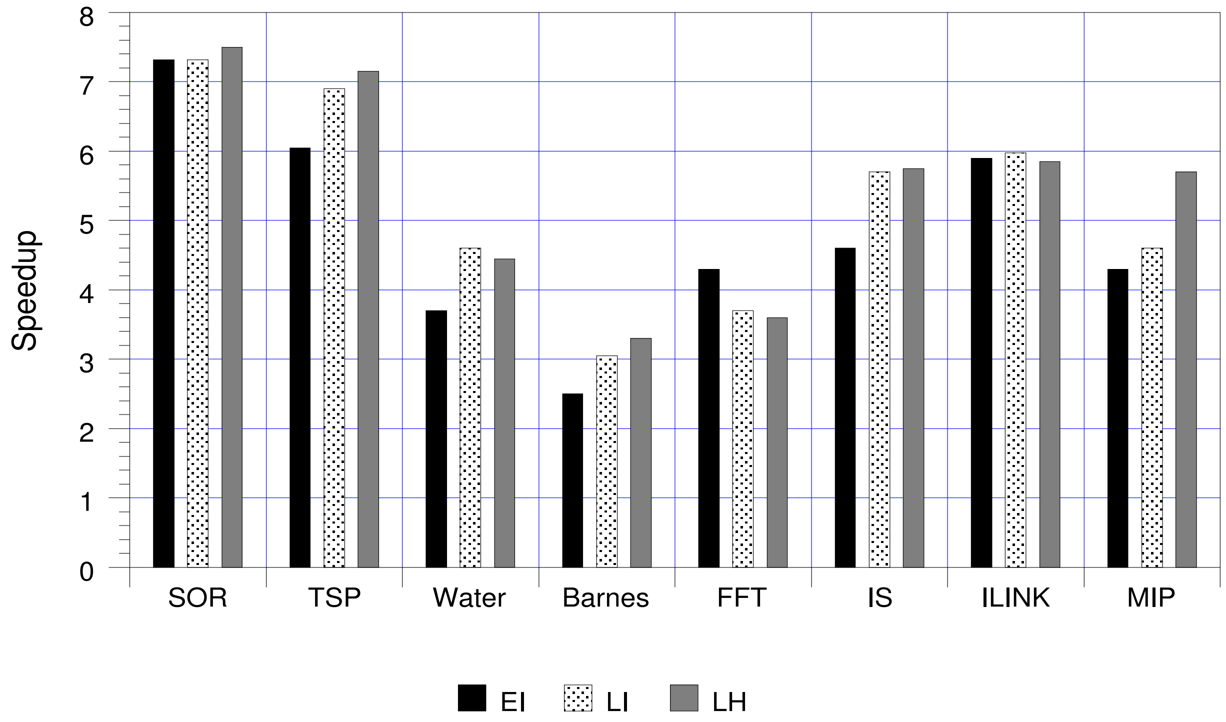

Figure 2 8-processor Speedups for EI, LI, and LH

Program	Prot	Run Time (secs)	Total Msgs (per sec)	Data (Kbytes per sec)	Access Misses (per sec)	Diffs Created (per sec)
SOR	EI	6.45	966.0	1419.2	66.9	0.0
	LI	6.24	790.9	1130.2	66.4	67.3
	LH	6.19	773.7	1170.7	0.0	93.1
TSP	EI	49.12	689.6	947.0	238.5	0.5
	LI	44.09	436.5	127.7	185.3	100.5
	LH	42.61	413.1	135.0	167.9	104.4
Water	EI	12.84	3822.0	1491.3	273.4	6.5
	LI	10.63	3127.4	836.0	313.2	224.1
	LH	10.86	2843.6	837.2	162.3	244.4
Barnes	EI	27.91	1063.1	295.9	435.4	212.0
	LI	22.56	2481.1	167.2	304.5	50.5
	LH	20.54	1051.1	187.0	237.5	54.4
FFT	EI	10.10	1011.7	3429.9	422.9	0.0
	LI	11.95	844.7	3375.7	355.4	528.7
	LH	11.86	876.7	4660.2	320.6	960.2
IS	EI	2.19	837.9	280.8	283.1	0.0
	LI	1.75	674.9	213.1	85.1	46.3
	LH	1.73	853.8	215.6	8.1	51.4
ILINK	EI	1030.8	472.4	571.7	118.3	16.9
	LI	1021.4	308.4	180.9	115.0	35.6
	LH	1027.5	134.6	201.4	79.5	38.2
MIP	EI	26.10	1781.0	1118.2	267.4	7.5
	LI	23.91	989.1	92.4	168.5	100.6
	LH	19.09	988.8	107.0	109.7	139.3

Table 3 EI, LI, and LH Rate Statistics

4.1.1 SOR

Our Successive Over-Relaxation (SOR) uses a simple iterative relaxation algorithm. The input is a two-dimensional grid. During each iteration, every matrix element is updated to the average of the four neighboring elements. To avoid overwriting an element before neighbors use it for their computations, we use a "red-black" approach, wherein every other element is updated during the first half-iteration, and the rest of the elements are updated during the second half-iteration. The work is parallelized by assigning a contiguous chunk of rows to each processor. Exchange of data between processors is therefore limited to those pages containing rows on the edge of the chunks. Barriers are used to synchronize all processors at the end of each half-iteration.

LI creates 25% fewer diffs than the other protocols because of an advantageous data layout and the fact that diffs are only created upon request. Under LH, neighboring processes exchange diffs via updates sent before arriving at a barrier. The primary advantage of the early updates is that they are unreliable, and so require only a single message. The access misses that occur in the absence of hybrid updates require at least two messages to handle. However, the gain in message handling overhead is partially offset by the cost of creating more diffs than LI.

LH's performance is also occasionally reduced by access misses that occur when update messages are either lost or delayed. The resulting message exchange not only slows down the processors involved, but also slows down the entire computation at the next barrier because of load imbalance.

EI requires more messages than the lazy protocols because each processor sends invalidates directly to other processors rather than appending them to barrier messages.

4.1.2 TSP

The Traveling Salesman Problem uses a branch-and-bound algorithm to find the minimum cost path that starts at a designated city, passes through every other city exactly once, and returns to the original city. Such a path is termed a *tour*. We assume a fully connected map of cities, and passage between each pair of cities has an associated weight. The cost of a tour is the sum of the weights of each leg of the tour. We solve a 19-city tour.

TSP processes synchronize entirely through locks. Like SOR, TSP has a very high computation to communication ratio, resulting in near-linear speedup. Therefore the lazy protocol's reduction in message traffic does not greatly affect overall performance.

The vast majority of messages in TSP are diff request and response messages, some of which are unnecessary given sufficient semantic information. The data accessed is the set of tour records used to hold path information while recursing. Tour records are often reused for different computations, and hence the previous contents are often not needed when a tour record is retrieved from the tour heap. The DSM system obliviously reconstructs the last contents of each accessed tour record even if application semantics do not require it.

A second source of overhead in TSP is contention for the centralized tour queue. Each thread performs a fairly extensive computation before releasing the tour queue, resulting in an average latency of acquiring the tour lock of over 22 milliseconds.

Despite these impediments, both of the lazy protocols achieve near-linear speedups, approximately 15% better than EI. Two factors cause the disparity between the lazy and eager protocols. First, EI suffers approximately one third more access misses because invalidates are propagated globally at release time, whereas invalidates propagate more slowly under the lazy protocols. Second, EI transfers nearly eight times as much data because invalidates always require complete pages to be fetched, while the lazy protocols usually require only a small number of diffs.

TSP performs only marginally better under LH than under LI. TSP has poor data locality, and therefore past behavior is not a good indicator of future access patterns. Nevertheless, LH sends approximately 9% fewer messages than LI for the 19 city problem, while sending only slightly more data.

4.1.3 Water

Water is a molecular dynamics simulation. Each time-step, the intra- and inter-molecular forces incident on a molecule are computed. In order to avoid an $\frac{n^2}{2}$ behavior, only molecules within half the box length of a given molecule are assumed to affect the molecule. We simulated 343 molecules for 5 steps.

The main shared data structure in Water is a large, one-dimensional array of molecules. Equal contiguous chunks of the array are assigned to each processor. Each molecule is represented by a 600-byte data structure that includes data describing the molecule's displacement, the first six derivatives, and computed forces.

Water has far higher communication requirements than the other applications, and under the lazy protocols almost 70% of this communication is lock requests and responses.

LI performs slightly better than LH because it creates fewer diffs. There is considerable false sharing because almost seven molecules fit on a single page. Since diffs are created very late under LI, the frequency of multiple molecule interactions being summarized by a single diff is higher than with LH.

EI again performs approximately 15% worse than the lazy protocols because of the need to fetch entire pages on access misses. While incurring a similar number of access misses, EI moves more than twice as much data.

4.1.4 Barnes-Hut

Barnes-Hut simulates the evolution of a system of bodies under the influence of gravitational forces. It is a classical gravitational N-body simulation, in which every body is modeled as a point mass and exerts forces on all other bodies in the system. If all pairwise forces are calculated directly, this has a complexity of $O(n^2)$ in the number of bodies, which is impractical for simulating large systems. Barnes-Hut is a hierarchical tree-based method that reduces the complexity to O(n log n). The program uses both locks and barriers for synchronization. We present results for a run using 4096 bodies.

The performance of this application is poor for all protocols because of the high synchronization rate and degree of false sharing. Nearly 98% of the messages under LI are diff messages. Not only does the high rate of access misses create overhead directly, but it contributes to load imbalance at barriers. From one barrier to the next, access misses and diff requests served vary significantly by process, and the number of access misses taken and diff requests served by a process correlates highly with the amount of time other processes have to wait at barriers. Overall, an average barrier takes almost 400 milliseconds for this application, while a null eight processor barrier takes slightly more than two milliseconds.

The use of LH's updates reduces the overall number of diffs requested by more than half. However, Table 3 shows that LH reduces access misses by only 22% from LI. Many of Barnes' access misses require more than a single diff in order to bring the page up to date. LH often eliminates some, but not all, of the diff requests for a given miss. Since misses requiring a single diff cost only marginally less than misses requiring multiple diffs, LH's impact on overall performance is less than might be expected. Nevertheless, it performs 10% better than LI, and nearly 35% better than EI.

EI's performance is seemingly an anomaly in that it uses less than half as many messages as LI and creates five times as many diffs. The explanation for this behavior is that our implementation resorts to updates, occasionally even global updates, to arbitrate multiple simultaneous invalidations of the same page. This complexity arises from the need to ensure that at least one valid copy of each page always survives. For Barnes, the result is that the EI's arbitration mechanism mimics LH's update mechanism at least part of the time.

4.1.5 FFT

This benchmark numerically solves a partial differential equation using forward and inverse FFT's. Assuming the input array is a $n_1 \times n_2 \times n_3$ array, A, organized in row-major order, we distribute the array elements along the first dimension of A, that is for any i, all elements of $A[i,*,*]$ are assigned to a single processor. A 1-D FFT is first performed on the $n_1 \times n_2$

n_3-point vectors, and then on the $n_2 \times n_3$ n_2-point vectors and each processor can work on its part of the array without any communication. Only when a processor is ready to work on the $n_1 - point$ vectors in the first dimension does it need to get the data from other processors. This means that only one transpose is needed for each iteration of the 3-D FFT.

When N processors are working in parallel, for every transpose each processor needs to send 1/N of its data to every other processor and receive 1/N of its data from each of the other processors. The array is often 1MB or larger, so the time spent doing the transpose is very large. The program uses only barriers for synchronization. We ran the tests with array dimensions of 64x64x32.

FFT is trivially parallelizable, but gets relatively poor speedup because of the low $O(\log n)$ computation to communication ratio. Processes running FFT communicate more than twice as much data per second than any other application.

This application illustrates a weakness of the lazy protocols. LI and LH create diffs describing each modification because every page of data is replicated over the course of the execution. However, in FFT a page is completely overwritten almost every time it is touched. Therefore, creating and applying a diff describing a changed page is less efficient *for this application* than merely sending the new page.

Under EI, access misses are handled by merely retrieving a copy of the page from another process, adding no additional diff creation/application overhead and not sending any extra data.

4.1.6 IS

This benchmark ranks an unsorted sequence of N keys. The *rank* of a key in a sequence is the index value i that the key would have if the sequence of keys were sorted. All the keys are integers in the range $[0,\ B_{max}]$ and the method used is counting, or bucket sort. The amount of computation required for this benchmark is relatively small – linear in the size of the array N. The amount of communication is proportional to the size of the key range, since an array of size B_{max} has to be passed around between processors. In the original benchmark specification, values for N and B_{max} are 2^{23} and 2^{19} respectively. Since this exceeds the amount of memory that we had available, we reduced these parameters to 2^{20} and 2^{7} respectively.

During a ranking, processes use a lock to acquire write permission to shared data. However, some of the shared memory is also read outside the locks. This results in extra invalidation messages for EI because each time a lock is released, invalidations are performed globally, even to those processes that only falsely share the modified pages. These extra messages do not occur under the lazy protocols because invalidations are only carried by synchronization messages, and the processes that are reading the shared data are doing so outside of any synchronization.

Table 3 shows that LH sends more messages than LI. The extra messages are barrier flushes that are often useless because many of the diffs communicated by the flushes have already been received via lock grant messages.

4.1.7 ILINK

Genetic linkage analysis is a statistical technique that uses family pedigree information to map human genes and locate disease genes in the human genome.

Our program is a parallel version of ILINK, which is part of the standard LINKAGE package for carrying out linkage analysis. ILINK searches for a maximum likelihood estimate of the multi-locus vector of *recombination probabilities* of several genes [12]. Given a fixed value of the recombination vector, the outer loops of the likelihood evaluation iterate over all the pedigrees and each nuclear family (consisting of parents and child) within each pedigree to update the probabilities of each genotype (see [7]) for each individual, which is stored in an array `genarray`.

A straightforward method of parallelizing this program is to split the iteration space among the processes and surround each addition with a lock to do it in place. This approach was deemed far too expensive either on a shared memory multiprocessor or on a DSM. Our version therefore uses a local copy of genarray to temporarily hold updates to the global array. They are eventually merged into the final copy after a barrier synchronization. ILINK's input consists of data on 12 families with autosomal dominant nonsyndromic cleft lip and palate (CLP).

LH is once again able to reduce the number of remote misses, thereby improving performance despite sending more data. The eager protocol does the worst because of the larger number of messages and data (entire pages are sent instead of diffs).

ILINK achieves less than linear speedup because of a combination of poor load balancing (this problem is inherent to the algorithm [7]) and sections of code that are executed serially. Consequently, speedups are somewhat lower than one would expect based on the communication and synchronization rates.

4.1.8 MIP

Mixed integer programming (MIP) is a version of linear programming where some or all of the variables are constrained to have an integer value, or sometimes to just the value 0 or 1. A wide variety of real-life problems can be expressed as MIP models, e.g., airline crew scheduling, network configuration, and plant design. MIP is hard not only in the standard technical sense, that is, "NP-hard," but it is also hard in the practical sense: real models regularly produce problem instances that cannot currently be solved.

The MIP code we use takes a *branch-and-cut* approach. The integer problem is first relaxed to a linear programming problem. This will in general lead to a solution in which some of the integer variables take on non-integer values. The next step is to pick one of these variables, and branch off two new linear programming problems, one with the added constraint that $x_i = \lfloor x_i \rfloor$ (the down branch) and another with the added constraint that $x_i = \lceil x_i \rceil$ (the up branch). Over time, the algorithm generates a tree of such branches. As soon as a solution is found, this solution establishes a *bound* on the final result. Nodes in the branch tree for which the solution of the LP problem generates a result that is inferior to this bound need not be explored any further. Additional techniques are used to speed up the algorithm, such as cutting planes, tighter linear constraints derived from the original constraints, and *plunging*, a depth-first search down the tree to find an integer solution and establish a bound as quickly as possible.

MIP is a work-queue based program implemented using locks. Hence, this program performs better on the lazy protocols, which are able to significantly reduce the number of messages and amount of data communicated. LH performs the best because of its ability to reduce the number of remote misses without significantly increasing the amount of data sent across the network.

4.2 Execution Time Breakdown

We used `qpt` [3] to break the execution time into several categories. Figure 3 shows the breakdown for each of our applications running on 8 processors under each of the protocols. The "Computation" category is the time spent executing application code; "Unix" is the time spent executing Unix system calls and library code (almost entirely time spent in Unix communication primitives); and "TreadMarks" is the time spent executing code in the TreadMarks library. "Idle Time" consists of several components, primarily time spent waiting for remote communication to complete and time wasted at barriers due to load imbalance.

The largest overhead components are the Unix and idle times. With the exception of ILINK, which has significant load imbalance, idle time is primarily time spent waiting for communication primitives to be executed by other processes. Hence, for all programs except ILINK, the sum of Unix and idle times is almost pure communication overhead. TreadMarks overhead, which includes time spent constructing twins and diffs, as well as applying the diffs, is much smaller than the communication overhead. We conclude that, *for this environment*, the complexity of the protocol matters far less than the number and size of messages required

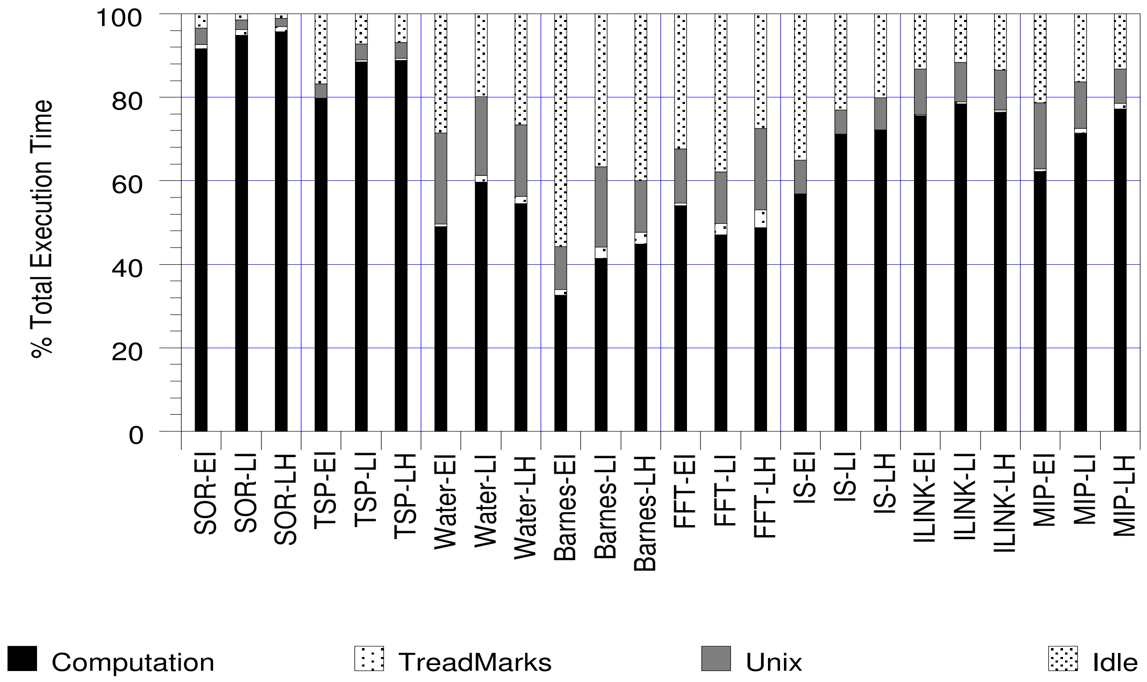

Figure 3 TreadMarks Execution Time Breakdown

to support the DSM environment.

Figure 4 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. For most of the programs, the eager protocol has the largest Unix communication overhead.

Figure 5 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 by user level routines to detect and capture changes to shared

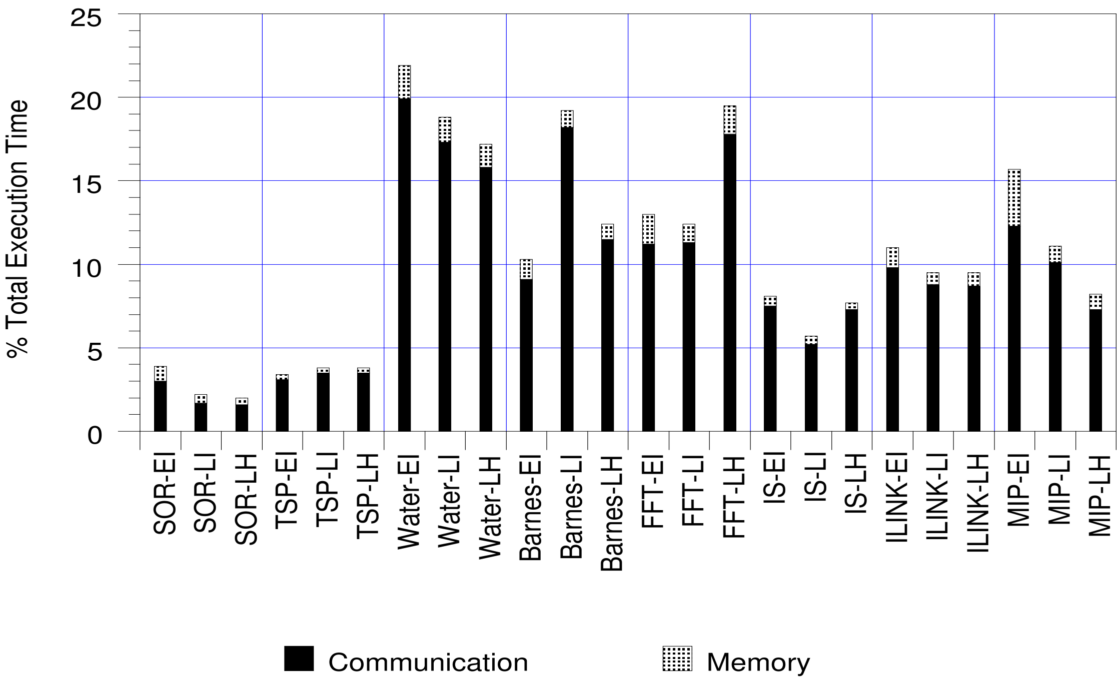

Figure 4 Unix Overhead Breakdown

pages. This includes twin and diff creation and diff application. "Consistency" is the time spent propagating and handling consistency information. "Other" consists primarily of time spent handling communication and synchronization. TreadMarks overhead is dominated by the memory management operations. The eager protocol has the least protocol overhead for all of the applications, indicating a trade-off between communication and protocol overhead. However, maintaining the rather complex partial ordering between intervals required by the lazy protocols adds only a small amount to the execution time.

4.3 Memory Overhead

All software DSMs trade memory for runtime overhead by replicating shared pages. However, lazy RC protocols also require significant amounts of memory to store diffs and consistency information. Diffs must be retained until they have been applied to every existing copy

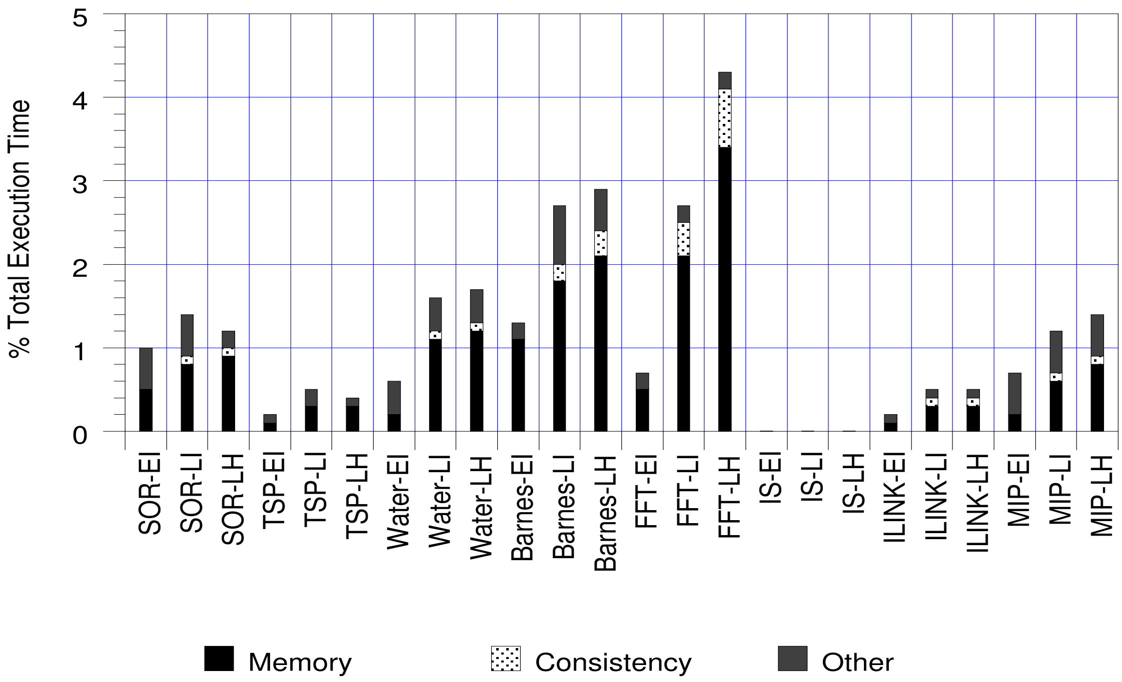

Figure 5 TreadMarks Overhead Breakdown

of the corresponding page. Rather than continuously evaluating this relatively complicated predicate, our implementations garbage-collect diff and consistency information only when internal buffers reach high-water marks.

Table 4 presents memory and garbage-collection overheads for the application suite with each of the lazy protocols. "Shared Size" is the size of the shared memory space, "Twin Space" is the maximum memory used by twins at all 8 processors, and "Diff Rate" is the rate at which diff storage is consumed. Nodes in our system each allocate approximately 4.5 MBytes of memory for diff and consistency information storage. "% Time GC" represents the percent of execution time spent garbage collecting. Since our algorithms update each modified page rather than ensuring only that a single copy of each page survives, this column captures the entire effect of garbage collection on the computation.

None of the applications spend more than 3.2% of their time garbage collecting, and most

Program		Shared Size (Kbytes)	Twin Space (Kbytes)	Diff Rate (Kbytes/sec)	% Time GC
SOR	LI	8040	7004	7.6	0
	LH	8040	7004	19.3	0
TSP	LI	3204	1096	8.0	0
	LH	3204	1068	8.7	0
Water	LI	238	392	180.6	0
	LH	238	432	446.9	0
Barnes	LI	690	2240	488.7	3.2
	LH	690	2236	615.2	1.2
FFT	LI	6291	5600	1304.0	0.4
	LH	6291	4060	1905.3	0.8
IS	LI	0.5	32	25.7	0
	LH	0.5	32	195.1	0
ILINK	LI	14670	1088	131.2	0.3
	LH	14670	1088	134.2	0.1
MIP	LI	150	316	3.3	0
	LH	150	352	3.6	0

Table 4 Memory Usage and GC Overhead

spend far less. The applications produce diffs at widely varying rates, with FFT creating enough diffs to completely overwrite its shared address space every three seconds.

EI discards diffs as soon as they are created because updates are performed globally, rather than locally as in the lazy protocols. "Twin Space" overhead for EI is similar to that of LI.

5 Simulation

Both networking hardware and operating system software affect the performance of application programs. A limitation of our empirical comparison is that the hardware and operating system costs are fixed. This section explores the relationship between the different consistency algorithms and protocols as the processor, network and operating system vary in speed.

5.1 Simulation Methodology

Our primary concern in selecting a simulation methodology was the ability to model accurately the software costs incurred by the different protocols. Therefore, we chose a method that allowed the execution of the actual protocol code on the simulator.

To meet our objectives, we use `vt` [3], a profiling tool that rewrites executable programs to incorporate instrumentation code that produces an estimated processor cycle count. To account for the time spent in the operating system handling page faults or passing messages, for example, we link the program to a library that intercepts system calls, and adds a specified number of cycles to the processor's counter. For message passing system calls, the library additionally computes the wire time for the message, based on the network speed and the message size. To arrive at the execution time on multiple processors, the library piggybacks a processor's cycle count on its synchronization messages, and adjusts the synchronizing processors' clocks according to the following rules: For a lock, the processor acquiring the lock must have a cycle count greater than that when the lock was last released plus communication latency; and, for a barrier, the processors departing from the barrier must have cycle counts greater than the highest cycle count among the processors arriving at the barrier.

In all cases, we simulate a switched LAN similar to an ATM LAN. We account for contention for each point-to-point link, that is, we simulate the serialization of messages requiring access to the same link.

To validate the simulator, we compared our model's simulated speedups to actual speedups for 8 processors on the different applications. In all cases, simulated speedup, the number of messages, and the total amount of data communicated came to within 10% of the measured counts.

Again, speedups were calculated with reference to single-threaded executions with TreadMarks library calls removed.

5.2 Effect of Communication Software Speed

The results of Section 4.2 suggest that reducing the cost of the communication software should improve performance. The cost has two components: a fixed, per message cost, and a per byte cost that accounts for the handling of different size messages. Figure 6 shows the simulated performance of an ATM network varying the fixed cost software overhead in the 8 processor case at a per byte software cost of 7 cycles/byte. All the applications presented have improved speedups with a zero fixed cost per message. The large speedups indicate the performance potential for the protocols, and the potential gains to be had from hardware support for message passing.

Low fixed costs favor protocols that send less data over those that send fewer messages. Therefore, LH, which reduces the number of access misses at the expense of sending slightly more data, loses ground to the other protocols as the fixed cost drops. EI also gains ground relative to LI because it sends many small messages at synchronization releases, whereas the lazy protocols send large messages at access misses. Overall, however, LI performs better than the other protocols because it sends less data.

Figure 7 presents the effect on speedup of varying the per byte software cost at a fixed cost of 5000 cycles per message. The increase in performance as per byte cost decreases is not as dramatic as when the per message cost drops because per message costs tend to dominate overall communication costs. The primary exception is FFT, which sends far more data than the other programs.

With two exceptions, the relative performance of the protocols changes only slightly. For Barnes, LI sends several times as many messages as the other protocols because it takes a large number of access misses. Both of the other protocols eliminate most of the access misses through updates. As per byte costs decrease, per message cost becomes more important, and LI's performance decreases with respect to the other protocols.

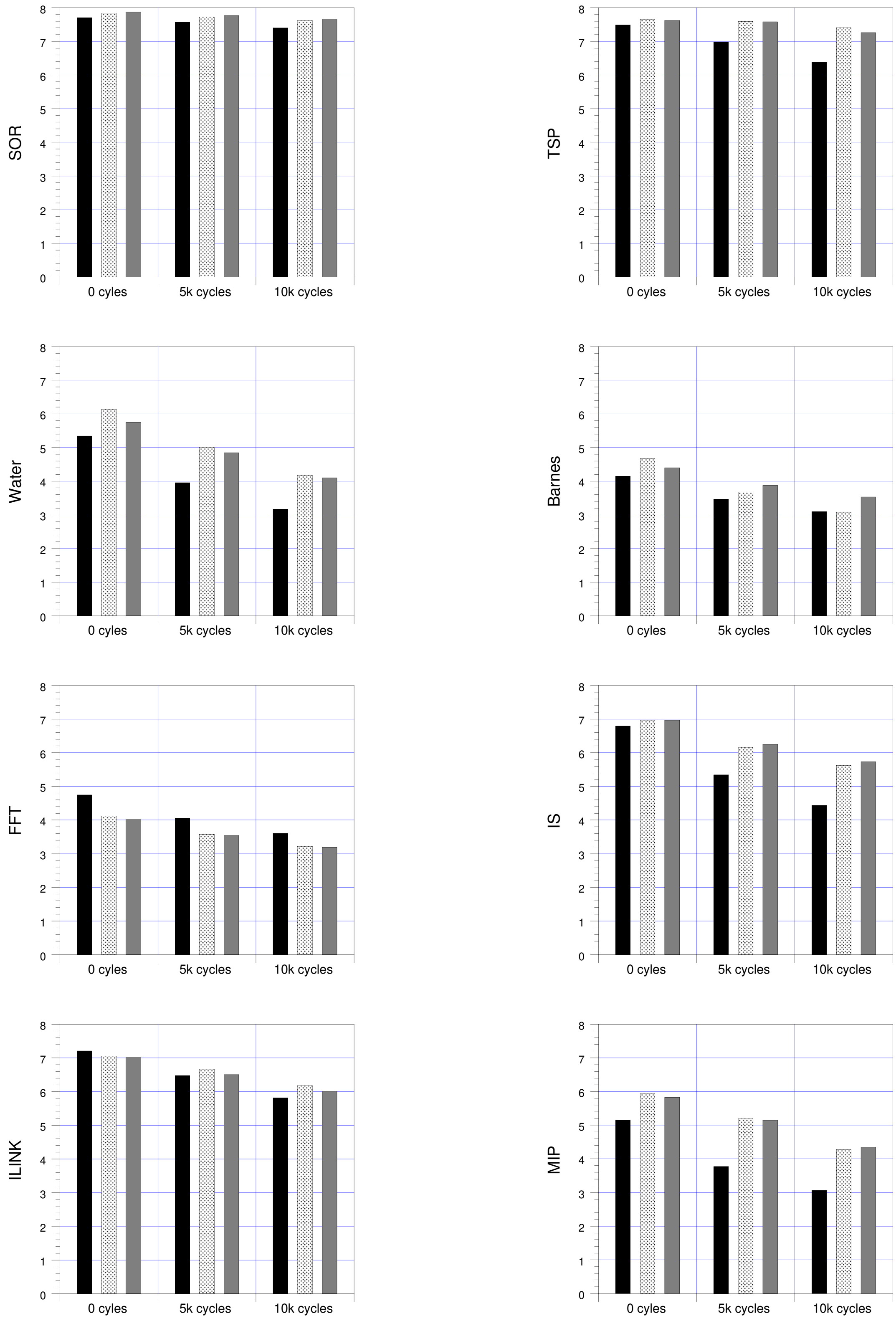

Figure 6 8-Processor Speedup Varying Fixed Message Cost (EI - black, LI - light gray, LH - dark gray)

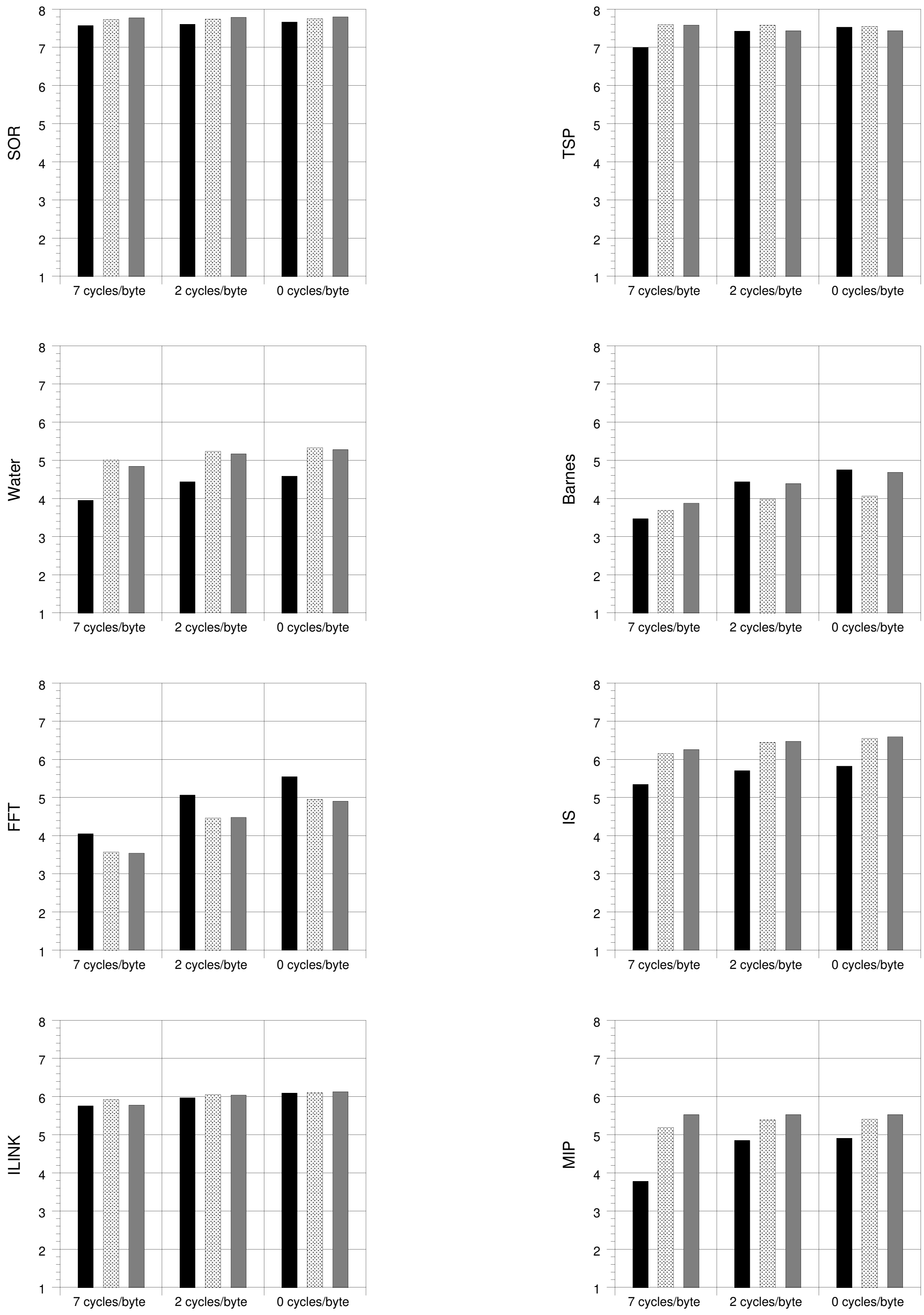

Figure 7 8-Processor Speedup Varying Per Byte Cost (EI - black, LI - light gray, LH - dark gray)

5.3 Effect of Network Speed

Access to the communication medium is a prime candidate for a bottleneck in any distributed system. Therefore, this section examines the effects of bandwidth variation.

Figure 8 summarizes changes in speedup for the programs when we vary bandwidth per link from 10 Mbits/sec to 1 Gigabit/sec, keeping the per byte software cost at 7 cycles/byte and the per message software cost at 5000 cycles. The performance difference between the programs from 10 to 100 Mbits/sec per link is much larger than the difference between 100 Mbits/secs and 1 Gigabit/sec. This is because many of the programs are bandwidth-limited at 10 Mbits/sec, but not at the higher speeds. Software overhead dominates at the higher data rates.

At a bandwidth of 10 Mbits/sec, LI outperforms the other protocols for nearly all of the applications because it sends less data than the other protocols. Since EI sends more data than either of the others, its performance is reduced proportionately.

FFT is an exception to the above generalized results. EI performs the best for FFT regardless of bandwidth or per message costs. The reason is that EI does not create diffs or twins at barriers, but instead migrates entire pages. Since pages are completely overwritten in each phase, the lazy protocols send at least as much data as EI, while still paying the overhead of creating and applying the diffs.

6 Related Work

RC was first proposed in the context of the DASH project [8]. In DASH, RC is implemented in hardware, using an invalidate protocol on a cache line basis. Given the small size of the cache line, false sharing is less of an issue, and a single-writer protocol is used.

The first software implementation of RC was carried out in the Munin systems [5]. Munin also introduced the notion of a multiple-writer protocol to combat false sharing. Munin allowed a number of protocols to be used, but the primary protocol was an eager update

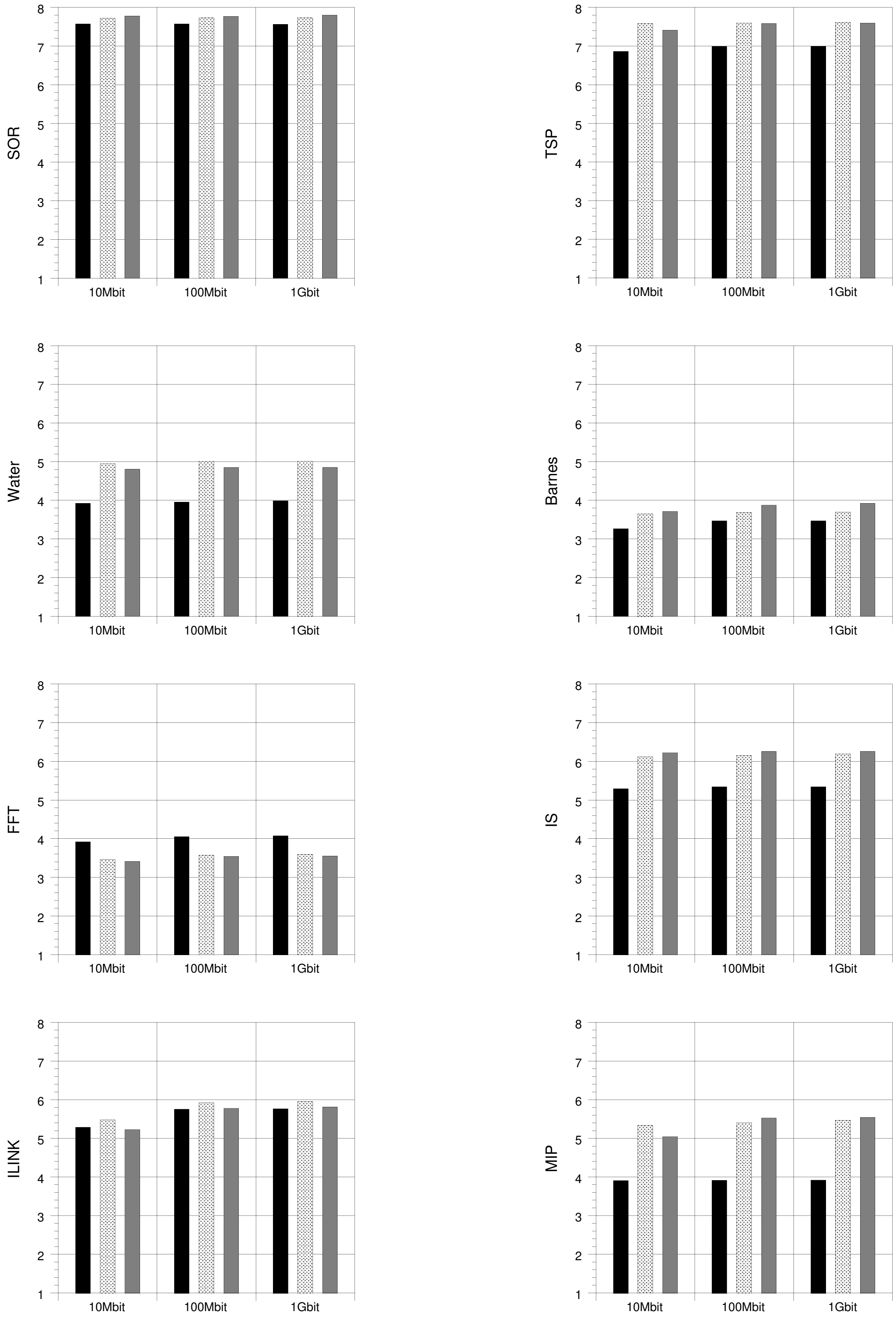

Figure 8 8-Processor Speedup Varying Bandwidth (EI - black, LI - light gray, LH - dark gray)

implementation of release consistency. Later work [6] has shown that the performance of EI and eager update are comparable.

Lazy release consistency was introduced in the TreadMarks system [10]. The default protocol in TreadMarks is LI, although Dwarkadas et al. [6] present simulation results for LH. Our work improves on earlier comparisons of various software implementations of RC by comparing actual implementations on the same platform, and by using measurements from these systems to validate simulation results that vary various environment parameters.

An interesting alternative to RC 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 over SC. Bershad et al. [4] also use a different strategy to implement EC in the Midway DSM system. Instead of relying on the VM system to detect shared memory updates, they modify the compiler to update software dirty bits.

7 Conclusions

In this paper, we have assessed the performance trade-offs between three different implementations of release consistency - an eager invalidate protocol, a lazy invalidate protocol, and a lazy hybrid protocol.

The protocols each have different strengths. The eager invalidate is less complex, but sends more messages and suffers more remote misses. At the cost of somewhat increased protocol complexity and overhead, the lazy invalidate protocol reduces remote misses and uses fewer messages. The hybrid protocol reduces remote misses even further, but sends more data and has the largest lock acquisition latency of any of the protocols.

We implemented each of these protocols in TreadMarks, a *distributed shared memory*

(DSM) system for standard Unix systems. Our hardware is a network of 8 DECstation-5000/240's that are connected by a 100-Mbps switch-based ATM LAN.

Our evaluation shows that the reduction in communication costs for the lazy protocols normally outweighs the decreased protocol complexity of EI on our experimental platform. The primary cause is the high per message and per byte communication cost of Unix software, which dominates the memory management and consistency overhead for all three protocols evaluated. No application spent more than 5% of its time executing protocol code, and in most cases much less time was spent. On average, the lazy hybrid protocol performs the best of the three protocols. On the 100-Mbps ATM LAN, the lazy hybrid achieves speedups of 7.5 for SOR, 7.2 for TSP, 5.8 for ILINK, 5.7 for IS, 5.7 for MIP, 4.5 for Water, 3.6 for FFT, and 3.3 for Barnes.

The relative performance of the protocols is dependent on the performance of the network, processor, and communication software. The gap between EI and the lazy protocols becomes larger as bandwidth decreases or software overhead increases. The relative performance of the two lazy protocols is more stable, but LI is favored as per message cost or bandwidth decreases, and LH is favored when applications are not bandwidth limited and per message costs increase.

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] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

[3] T. Ball and J. Larus. Optimally profiling and tracing programs. In *POPL92*, pages 59–70, January 1992.

[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] 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.

[6] 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.

[7] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A.L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, 1994.

[8] 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.

[9] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency.* PhD thesis, Rice University, December 1994.

[10] 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.

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

[12] 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.

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

[14] E. L. Lusk and R. A. Overbeek et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc, 1987.

[15] 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.

Pete Keleher received the B.S., M.S., and Ph.D. degrees from Rice University in 1986, 1992, and 1995, respectively. He is currently on the faculty at the University of Maryland, College Park. His research interests include parallel and distributed systems, computer networks and architecture, and parallel languages. *e-mail:* keleher@cs.umd.edu.

Alan Cox received the B.S. degree from Carnegie Mellon University in 1986 and the M.S. and Ph.D. degrees from the University of Rochester in 1988 and 1992, respectively. He is currently on the faculty at Rice University. His research interests include parallel and distributed systems, distributed garbage collection, and multi-computer architectures. *e-mail:* alc@cs.rice.edu

Sandhya Dwarkadas received the B.Tech. degree from the Indian Institute of Technology, Madras, India, in 1986, and the M.S. and Ph.D. degrees from Rice University in 1989 and 1993. She is currently a research scientist at Rice University. Her research interests include parallel and distributed systems, parallel computer architecture, parallel computation, simulation methodology, and performance evaluation. *e-mail:* sandhya@cs.rice.edu

Willy Zwaenepoel received the B.S. degree from the University of Gent, Belgium, in 1979, and the M.S. and Ph.D. degrees from Stanford University in 1980 and 1984. Since 1984,

he has been on the faculty at Rice University. His research interests are in distributed operating systems and in parallel computation. While at Stanford, he worked on the first version of the V kernel, including work on group communication and remote file access performance. At Rice, he has worked on fault tolerance, protocol performance, optimistic computations, distributed shared memory, and nonvolatile memory. *e-mail:* willy@cs.rice.edu