# Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology

Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Zwaenepoel
Department of Computer Science
Rice University *

## Abstract

We evaluate the effect of processor speed, network characteristics, and software overhead on the performance of release-consistent software distributed shared memory. We examine five different protocols for implementing release consistency: eager update, eager invalidate, lazy update, lazy invalidate, and a new protocol called *lazy hybrid*. This lazy hybrid protocol combines the benefits of both lazy update and lazy invalidate.

Our simulations indicate that with the processors and networks that are becoming available, coarse-grained applications such as Jacobi and TSP perform well, more or less independent of the protocol used. Medium-grained applications, such as Water, can achieve good performance, but the choice of protocol is critical. For sixteen processors, the best protocol, lazy hybrid, performed more than three times better than the worst, the eager update. Fine-grained applications such as Cholesky achieve little speedup regardless of the protocol used because of the frequency of synchronization operations and the high latency involved.

While the use of relaxed memory models, lazy implementations, and multiple-writer protocols has reduced the impact of false sharing, synchronization latency remains a serious problem for software distributed shared memory systems. These results suggest that future work on software DSMs should concentrate on reducing the amount of synchronization or its effect.

## 1 Introduction

Although several models and algorithms for software distributed shared memory (DSM) have been published, performance reports have been relatively rare. The few performance results that have been published consist of measurements of a particular implementation in a particular hardware and software environment [3, 5, 6, 13]. Since the cost of communication is very important to the performance of a DSM, these results are highly sensitive to the implementation of the communication software. Furthermore, the hardware environments of many of these implementations are by now obsolete. Much faster processors are commonplace, and much faster networks are becoming available.

We are focusing on DSMs that support release consistency [9], i.e., where memory is guaranteed to be consistent only following certain synchronization operations. The goals of this paper are two-fold: (1) to gain an understanding of how the performance of release consistent software DSM depends on processor speed, network characteristics, and software overhead, and (2) to compare the performance of several protocols for supporting release consistency in a software DSM.

The evaluation is done by execution-driven simulation [7]. The application programs we use have been written for (hardware) shared memory multiprocessors. Our results may therefore be viewed as an indication of the possibility of "porting" shared memory programs to software DSMs, but it should be recognized that better results may be obtained by tuning the programs to a DSM environment. The application programs are Jacobi, Traveling Salesman Problem (TSP), and Water and Cholesky from the SPLASH benchmark suite [14]. Jacobi and TSP exhibit coarse-grained parallelism, with little synchronization relative to the amount of computation, whereas Water may be characterized as medium-grained, and Cholesky as fine-grained.

We find that, with current processors, the bandwidth of the 10-megabit Ethernet becomes a bottleneck, limiting the speedups even for a coarse-grained application such as Jacobi to about 5 on 16 processors. With a 100-megabit point-to-point network, representative of the ATM LANs now appearing on the market, we get good speedups even for small sizes of coarse-grained prob-

*This work was supported in part by NSF Grants CCR-9116343 and CCR-9211004, Texas ATP Grant No. 0036404013 and by a NASA Graduate Fellowship.

lems such as Jacobi and TSP, moderate speedups for Water, and very little speedup for Cholesky. Regardless of the considerable bandwidth available on these networks, Cholesky's performance is constrained by the very high number of synchronization operations.

Among the protocols for implementing software release consistency, we distinguish between *eager* and *lazy* protocols. *Eager* protocols push modifications to all cachers at synchronization variable releases [5]. In contrast, *lazy* protocols [11] pull the modifications at synchronization variable acquires, and communicate only with the acquirer. Both eager and lazy release consistency can be implemented using either invalidate or update protocols. We present a new *lazy hybrid* protocol that combines the benefits of update and invalidate: few access misses, low data and message counts, and low lock acquisition latency.

Our simulations indicate that the lazy algorithm and the hybrid protocol significantly improve the performance of medium-grained programs, those on the boundary of what can be supported efficiently by a software DSM. Communication in coarse-grained programs is sufficiently rare that the choice of protocols becomes less important. The eager algorithms perform slightly better for TSP because the branch-and-bound algorithm benefits from the early updates in the eager protocols (see Section 6.2). For the fine-grained programs, lazy release consistency and the hybrid protocol reduce the number of messages and the amount of data drastically, but the communication requirements are still beyond what can be supported efficiently on a software DSM. For these kinds of applications, techniques such as multithreading and code restructuring may prove useful.

The outline of the rest of this paper is as follows. Section 2 briefly reviews release consistency, and the eager and lazy implementation algorithms. Section 3 describes the hybrid protocol. Section 4 details the implementation of the protocols we simulated. Section 5 discusses our simulation methodology, and Section 6 presents the simulation results. We briefly survey related work in Section 7 and conclude in Section 8.

## 2 Release Consistency

For completeness, we reiterate in this section the main concepts behind release consistency (RC) [9], eager release consistency (ERC) [5], and lazy release consistency (LRC) [11].

RC [9] is a form of relaxed memory consistency that allows the effects of shared memory accesses to be delayed until selected synchronization accesses occur. Simplifying matters somewhat, shared memory accesses are labeled either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. Acquires and releases may be thought of as conventional synchronization operations on a lock, but other synchronization mechanisms can be mapped on to this model as well. Essentially, RC requires ordinary shared memory accesses to be performed only when a subsequent release by the same processor is performed. RC implementations can delay the effects of shared memory accesses as long as they meet this constraint.

For instance, the DASH [12] implementation of RC buffers and pipelines writes without blocking the processor. A subsequent release is not allowed to perform (i.e., the corresponding lock cannot be granted to another processor) until acknowledgments have been received for all outstanding invalidations. While this strategy masks latency, in a software implementation it is also important to reduce the *number* of messages sent because of the high per message cost.

In an *eager* software implementation of RC such as Munin's multiple-writer protocol [5], a processor delays propagating its modifications of shared data until it executes a release (see Figures 1 and 2). *Lazy* implementations of RC further delay the propagation of modifications until the acquire. At that time, the last releaser piggybacks a set of *write notices* on the lock grant message sent to the acquirer. These write notices describe the shared data modifications that precede the acquire according to the *happened-before-1* partial order [1]. The *happened-before-1* partial order is essentially the union of the total processor order of the memory accesses on each individual processor and the partial order of release-acquire pairs. The *happened-before-1* partial order can be represented efficiently by tagging write notices with vector timestamps [11]. At acquire time, the acquiring processor determines the pages for which the incoming write notices contain vector timestamps larger than the timestamp of its copy of that page in memory. For those pages, the shared data modifications described in the write notices must be reflected in the acquirer's copy either by invalidating or by updating that copy. The tradeoffs between invalidate and update and a new *hybrid* protocol are discussed in the next section.

## 3 A Hybrid Protocol for LRC

A lazy invalidate protocol invalidates the local copy of a page for which a write notice with a larger timestamp is received (see Figure 3). The lazy update protocol never invalidates pages to maintain consistency. Instead, acquiring processes retrieve all modifications named by

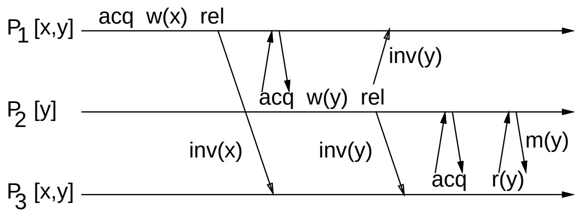


**Figure 1** Eager Invalidate

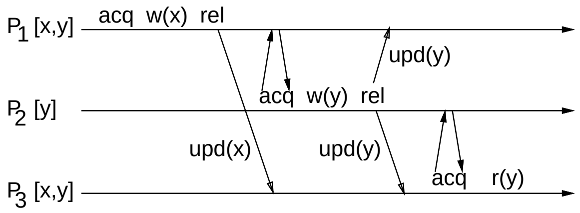


**Figure 2** Eager Update

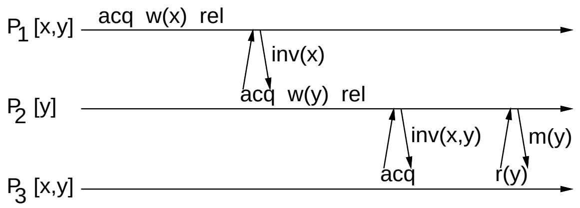


**Figure 3** Lazy Invalidate

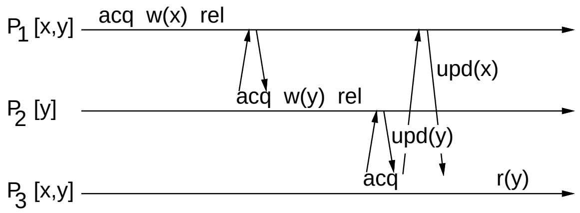


**Figure 4** Lazy Update

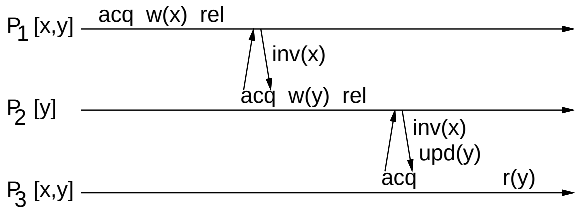


**Figure 5** Lazy Hybrid

incoming write notices for any page that is cached locally (see Figure 4). As an optimization, the releaser piggybacks the modifications it has available locally on the lock grant message.

In the lazy *hybrid* protocol, as in the lazy update protocol, the releaser piggybacks on the lock grant message, in addition to write notices, the modifications to those pages that it believes the acquirer has a copy of in its memory. However, unlike in the lazy update protocol, the acquirer does not make any attempt to obtain any other modifications. Instead, it invalidates the pages for which it received write notices but for which no modifications were included in the lock grant message.

Previous simulations [11] indicate that (1) the lazy protocols send fewer messages and less data than the eager protocols, and (2) the lazy update protocol send fewer messages in most cases than the lazy invalidate protocol, while the lazy invalidate protocol sends less data than the lazy update protocol. The reduction in the number of access misses outweighs the extra messages exchanged at the time of synchronization. Also, the reduced access misses result in reduced latency, thus favoring the update protocol.

However, the choice of a lazy or an eager algorithm, and furthermore the choice between an update or an invalidate protocol also affects the lock acquisition latency. We distinguish two cases.

1. The lock request is pending at the time of the release. The lazy invalidate protocol has the shortest lock acquisition latency, since a single message from the releaser to the acquirer suffices, followed by the invalidations at the acquirer, a purely local operation. In contrast, the eager algorithms must update or invalidate all other cachers of pages that have been modified at the releaser, and the lazy update protocol must retrieve all the modifications that precede the acquire, again potentially a multi-host operation.

2. The lock request is not yet pending at the time of the release. The eager algorithms have the lowest lock acquisition latency, followed closely by the lazy invalidate protocol. All require a single message exchange between the releaser and the acquirer, but the lazy invalidate protocol also needs to invalidate any local pages that have been modified. The lazy update protocol potentially requires a multi-host operation, resulting in higher lock acquisition latency.

The lazy hybrid protocol combines the advantages of lazy update and lazy invalidate protocols. First, like the invalidate protocol, the hybrid only exchanges a single pair of messages between the acquiring and the releasing processor. As a result, lock acquisition latency for the lazy hybrid protocol is close to that of the lazy invalidate protocol. The only additional overhead comes from the need to send and process the modifications piggybacked on the lock grant message. Second, the amount of data exchanged is smaller than for the update protocol. Finally, the hybrid sends updates for recently modified pages cached by the acquirer. It is likely that these pages will be accessed by the acquirer, thus reducing the number of access misses, and, as a result, reducing the latency and the number of miss messages.

# 4 Protocol Implementations

In this section we describe the details of the five protocols that we simulated: lazy hybrid (LH), lazy invalidate (LI), lazy update (LU), eager invalidate (EI), and eager update (EU).

All five are *multiple-writer* protocols. Multiple processors can concurrently write to their own copy of a page with their separate modifications being merged at a subsequent release, in accordance with the RC model. This contrasts with the exclusive-writer protocol used, for instance, in DASH [9], where a processor must obtain exclusive access to a cache line before it can be modified. Experience with Munin [5] indicates that multiple-writer protocols perform well in software DSMs, because they can handle false sharing without generating large amounts of message traffic between synchronization points.

All of the protocols support the use of exclusive locks and global barriers to synchronize access to shared memory. Processors acquire locks by sending a request to the statically assigned owner, who forwards the request on to the current holder of the lock. "Locks" and "unlocks" are mapped onto acquires and releases in a straightforward manner. Barriers are implemented using a *barrier master* that collects arrival messages and distributes departure messages. In terms of consistency information, a barrier arrival is modeled as a release, while a departure is modeled as an acquire on each of the other processors.

Processes exchange three types of information at locks and barriers: synchronization information, consistency information, and data. The consistency information is a collection of *write notices*, each of which contains the processor identification and the vector timestamp of the modification. Consistency information can be piggybacked on synchronization messages, but often the data comprising the modifications to shared memory can not. Most shared data exchanged in the protocols is in the form of *diffs*, which are runlength encodings of the modified data of a single page. Sending diffs instead of entire pages greatly reduces data traffic, and allows multiple concurrent modifications to be merged into a single version.

Each shared page has a unique, statically assigned owner. Each processor keeps an approximate copyset for every shared memory page. The copyset is initialized to the owner's copyset when a page is initially received, and updated according to subsequent write notices and diff requests. The copysets are used in the eager protocols to flush invalidations or updates to all other processors at releases. Since the copyset is only approximate, multiple rounds are sometimes needed to ensure that the consistency information reaches every cacher of the modified pages. The copysets are used by LH to determine which write notices should be accompanied by diffs.

Table 1 summarizes the message counts for locks, barriers, and access misses for each of the protocols. In this table, the *concurrent last modifiers* for a page are the processors that created modifications that do not precede, according to *happened-before-1*, any other known modifications to that page.

## 4.1 The Eager Protocols

### 4.1.1 Locks

We base our eager RC algorithms on Munin's multiple-writer protocol [5]. A processor delays propagating its modifications of shared data until it comes to a release. At that time, write notices, together with diffs in the EU protocol, are sent to all other processors that cache the modified pages, possibly taking multiple rounds if the local copysets are not up to date.

A lock release is delayed until all modifications have been acknowledged by the remote cachers. An acquire consists solely of locating the processor that executed the corresponding release and transferring the synchronization variable. No consistency-related operations occur at lock acquires.

### 4.1.2 Barriers

At barrier arrivals, the EI protocol sends synchronization and consistency information to the master in a single message. However, the EI barrier protocol has a slight complication in that multiple processors may invalidate the same page at a barrier. In order to prevent all copies of a page from being invalidated, the master designates one processor as the "winner" for each page. Only the winner retains a valid copy for a given concurrently modified page. The losers forward their modifications to the winner and invalidate their local copies.

In the EU protocol, each processor flushes modifications to all other cachers of locally modified pages before sending a synchronization message to the barrier master.

#### 4.1.3 Access Misses

Access misses are treated identically for both protocols. A message is sent to the owner of the page. The owner forwards the request to a processor that has a valid copy. This processor then sends the page to the processor that incurred the access miss.

### 4.2 The Lazy Protocols

#### 4.2.1 Locks

At an acquire, the protocol locates the processor that last executed a release on the same variable. The releaser sends both synchronization and consistency information to the acquirer in a single message. The consistency information consists of write notices for all modifications that have been performed at the releaser but not the acquirer. While LI moves data only in response to access misses, both the LH and LU protocols send diffs along with the synchronization and consistency information. However, LH moves diffs only from the releaser to the acquirer, and hence can append them to an already existing message. The releaser sends all diffs that correspond to modifications being performed at the acquire for the first time, such that for each diff the acquirer is in the releaser's copyset for the page named by the diff. Pages named by write notices that arrive without diffs are invalidated.

The LU protocol *never* invalidates pages. An acquire does not succeed until all of the diffs described by the new write notices have been obtained. In general, the acquirer must talk to other processors in order to pick up all of the required diffs. However, the number of processors with which the acquirer needs to communicate can be reduced because of the following observation. If processor $p$ modifies a page at time $t$, then all diffs of that page that precede the modification according to *happened-before-1* can be obtained from processor $p$.

#### 4.2.2 Barriers

At barrier arrivals, the LI protocol sends synchronization information and write notices to the master in a single message. When all processors have arrived, the barrier master sends a single message to each processors that contains the barrier release as well as all the write notices that it has collected.

LH and LU barrier arrivals are handled similarly. In both cases, each processor pushes updates to all processors that cache pages that have been modified locally, before sending a barrier arrival message to the master. The only difference is that in LU, the processes must wait on the arrival of the data before departing from the barrier.

#### 4.2.3 Access Misses

Access misses are handled identically by LH, LI, and LU. At a miss, a copy of the page and a number of diffs may have to be retrieved. The number of sites that need to be queried for diffs can be reduced through the same logic as in Section 4.2.1. The new diffs are then merged into the page and the processor is allowed to proceed. The lazy protocols determine the location of a page or updates to the page entirely on the basis of local information. No additional messages are required, unlike in other DSM systems [13].

## 5 Methodology

### 5.1 Application Suite

We simulated four programs, from three different classes of applications. Jacobi and TSP are coarse-grained programs with a large amount of computation relative to synchronization (323,840 and 18,092,000 cycles per processor between off-node synchronization operations, respectively, at 16 processors). Our Jacobi program is a simple Successive Over-Relaxation program that works on grids of 512 by 512 elements. TSP solves the traveling salesman problem for 18-city tours. Water, from the SPLASH suite[14], is a medium grained molecular dynamics simulation (19200 cycles per processor between off-node synchronization operations). We ran Water with the default parameters: 288 molecules for 2 steps. Cholesky performs parallel factorization of sparse positive definite matrices, and is an example of a program with fine-grained parallelism from the SPLASH benchmark suite (4,000 cycles per processor between off-node synchronization operations). Cholesky was run with the default input file, 'bcsstk14'. TSP and Cholesky use only locks for synchronization, Jacobi uses only barriers, and Water uses both.

### 5.2 Architectural Model

We used two basic architectural models, an *Ethernet* model and an *ATM* switch model. Both models assume 40MHz RISC processors with 64 Kbyte direct-mapped caches and a 12 cycle memory latency, 4096 byte pages, and an infinite local memory (no capacity misses). The ethernet is modeled as a 10 MBit/sec broadcast network, while the ATM is modeled as a 100 MBit/sec cross-bar switch.

### 5.3 Protocol Simulation

Each message exchanged by the protocols was modeled by the wire time consumed by sending the mes-

|  | Access Miss | Lock | Unlock | Barrier |
|---|---|---|---|---|
| LH | 2m | 3 | 0 | 2(n-1)+u |
| LI | 2m | 3 | 0 | 2(n-1) |
| LU | 2m | 3+2h | 0 | 2(n-1)+2u |
| EI | 2 or 3 | 3 | 2c | 2(n-1) + v |
| EU | 2 | 3 | 2c | 2(n-1) + 2u |

$m$ = # concurrent last modifiers for the missing page
$h$ = # other concurrent last modifiers for any local page
$c$ = # other cachers of the page
$n$ = # processors in system
$p$ = # pages in system
$u = \sum_{i=1}^{n}$(# other procs caching pages modified by $i$)
$v = \sum_{i=1}^{p}$(# excess invalidators of page $i$)

**Table 1** Shared Memory Operation Message Costs

sage, any inherent network latency, contention for the network, and a *software overhead* that represents the operating system cost of calling a user-level handler for incoming messages, creating and reading the messages in the DSM software, and the cost of the DSM protocol implementation. This cost is set at (1000 + message length $*$ 1.5/4) processor cycles at both the destination and source of each message. These figures were modeled after the Peregrine [10] implementation overheads. Peregrine is an RPC system that provides performance close to optimal by avoiding intermediate copying. The lazy implementation's extra complexity is modeled by doubling the per-byte message overhead both at the sender and at the receiver. Diffs are modeled by charging four cycles per word per page for each modified page at the time of diff creation. Although all messages are simulated, protocol-specific consistency information is not reflected in the amount of data sent. Only the actual shared data moved by the protocols is included in message lengths.

# 6 Simulation Results

## 6.1 DSM on an Ethernet

Although prior work [5] showed that Ethernet-based software DSMs can achieve significant speedups, we find that for modern processors the Ethernet is no longer a viable option. Figure 6 shows the speedup of Jacobi, a coarse-grained program. Jacobi's speedup peaks at 5.2 for eight processors, and declines rapidly thereafter. While Jacobi's communication needs are modest in comparison with other programs, the individual processors execute identical code and therefore create significant network contention at each barrier. This contention is especially significant for the update protocols, in which each processor sends updates to its neighbors prior to the barrier. In an 8-processor run, processors on average wait more than 3 milliseconds before gaining control of the Ethernet.

## 6.2 DSM on an ATM

The emerging ATM networks have several advantages over the Ethernet. Foremost among these are increased bandwidth and reduced opportunity for contention. Unlike the Ethernet, in which all processors seeking to communicate contend with each other, processors in an ATM network can communicate concurrently and interfere only when they try to send to a common destination.

Figures 7-9 summarize the performance of the Jacobi program on an ATM. While the Ethernet simulation of Jacobi achieved a speedup of about 5, the ATM version reaches 14. Part of this increase is due to the increased bandwidth, but much of it is due to the fact that no more than two competing updates (from each of a processor's two neighbors) ever arrive at a single destination during one interval. The performance of all five protocols is roughly the same for this program because of the regular nearest-neighbor sharing. The invalidate protocols fare slightly worse than the update protocols because pages on the edge of a processor's assigned data are invalidated at barriers, and have to be paged across the network. The lazy protocols perform slightly worse than the eager protocols because of the extra overhead

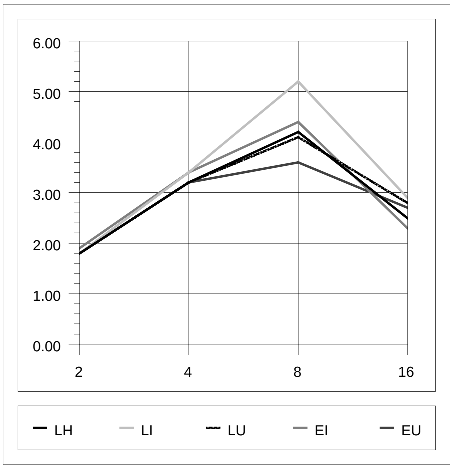


**Figure 6** Speedup for Jacobi on Ethernet

added in the simulation for message processing. This overhead is probably unjustified for Jacobi because of the nature of communication involved. As will be seen in all of the simulations, EI moves significantly more data than the other protocols because its access misses cause entire pages to be transmitted, rather than diffs.

Like Jacobi, TSP is a coarse-grained program with modest amounts of communication. Much of TSP's inefficiency results from contention for a global tour queue. Fully 10% of a 16-processor execution is wasted waiting for the queue lock. In order to prevent repeated acquires because of unpromising tours, each acquirer holds the queue's lock while making a preliminary check on the topmost tour. If the tour is promising, the queue's lock is released. Otherwise, the acquirer removes another tour from the queue.

Figures 10-12 present TSP's performance. There is little variation among the lazy protocols and among the eager protocols because of the large granularity and the contention for the queue lock. However, the speedup for the eager protocols is better than for the lazy protocols. TSP uses a branch-and-bound algorithm, using a global minimum to prune recursive searches. Read access to the current minimum is not synchronized. A processor may therefore read a stale version of the minimum. The lock protecting the minimum is acquired only when the length of the tour just explored is smaller than (the potentially stale value of) the minimum. The length is then rechecked against the value of the minimum, which is now guaranteed to be up to date, and the minimum is updated, if necessary. The eager protocols push out the new value of the minimum at each release, and therefore local copies of the minimum are frequently updated. It is thus unlikely that a processor would read a stale value, unlike with the lazy protocols where the local copy is only updated as a result of an acquire. Since the algorithm uses the global minimum to prune searches, such stale values may cause TSP to explore more unpromising tours with the lazy protocols.

Water is a medium-grained program that uses both locks and barriers. Water's data consists primarily of an array of molecules, each protected by a lock. During each iteration, the force vectors of all molecules with a spherical cutoff range of a molecule are updated to reflect the molecule's influence. In combination with the relatively small size of the molecule structure in comparison with the size of a page, this creates a large amount of false sharing. The simulation results for Water can be seen in Figures 13-15. LH performs better than the other protocols because the molecules' migratory behavior during the force modification phase allows the protocol to have far fewer cache misses, and hence messages, than the other protocols. The lazy protocols perform better than the eager protocols, and

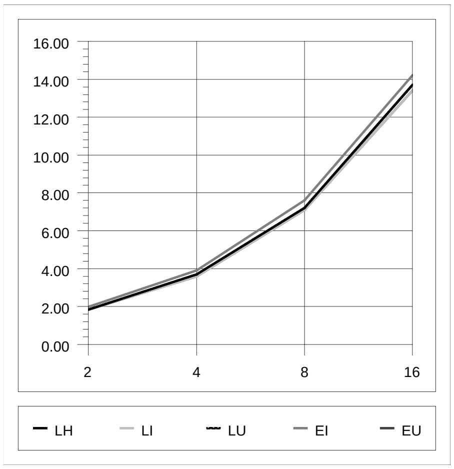


**Figure 7** Speedup for Jacobi

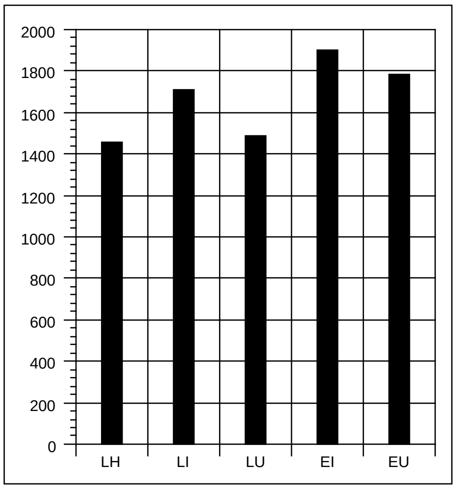


**Figure 8** Message Count in Jacobi

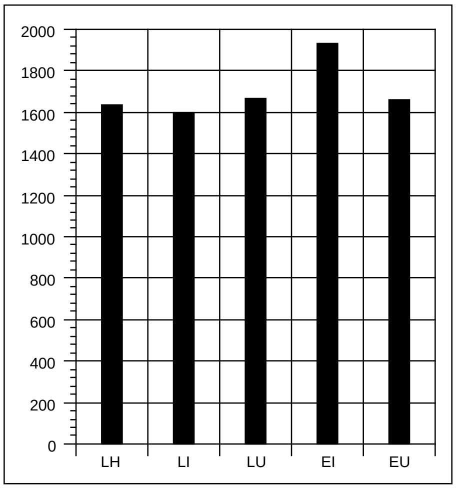


**Figure 9** Data (Kbytes) Transmitted in Jacobi

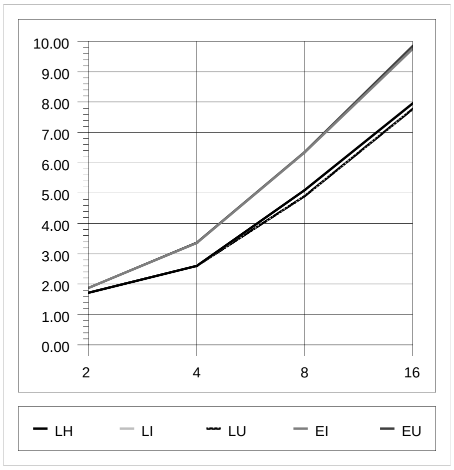


**Figure 10** Speedup for TSP

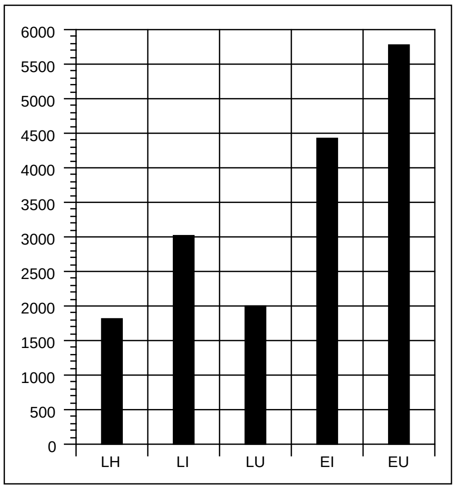


**Figure 11** Message Count in TSP

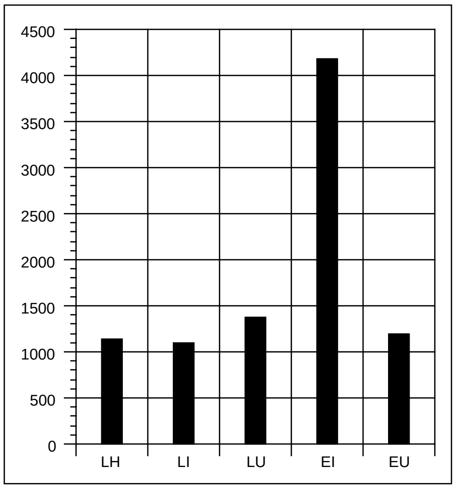


**Figure 12** Data (Kbytes) Transmitted in TSP

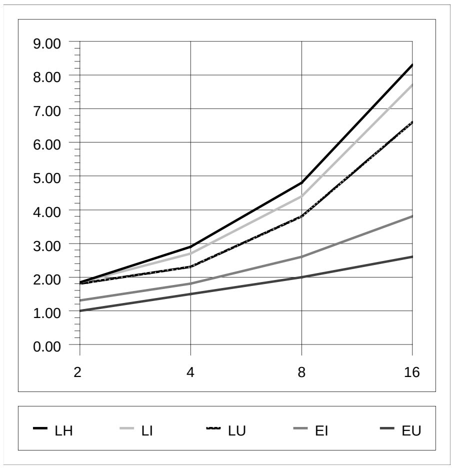


**Figure 13** Speedup for Water

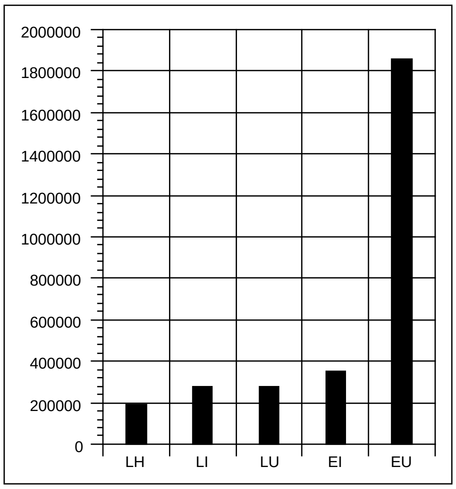


**Figure 14** Message Count in Water

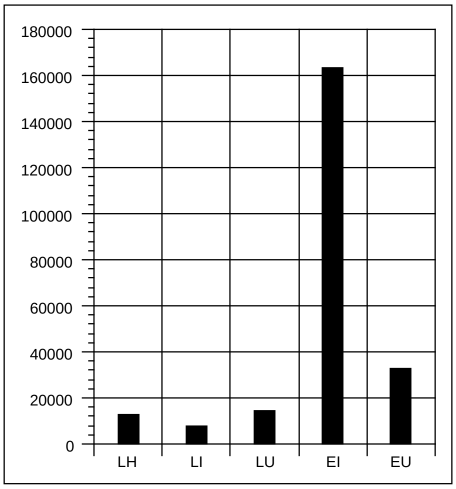


**Figure 15** Data (Kbytes) Transmitted in Water

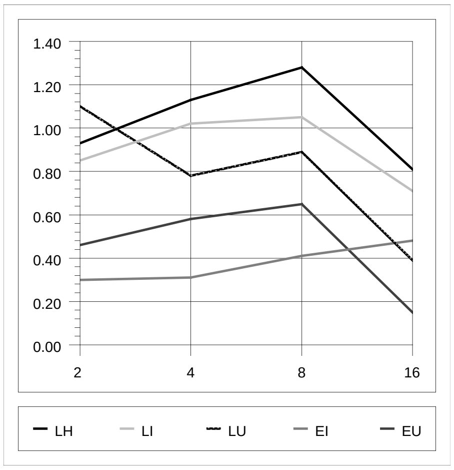


**Figure 16** Speedup for Cholesky

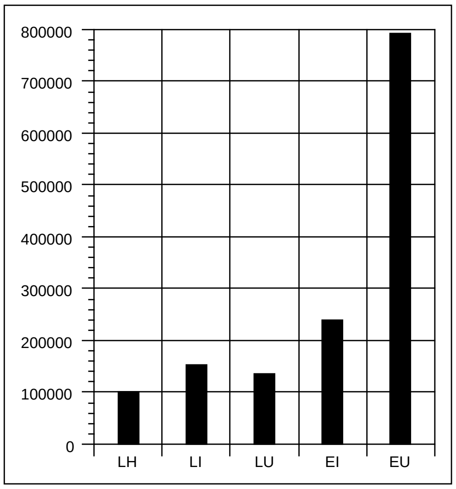


**Figure 17** Message Count in Cholesky

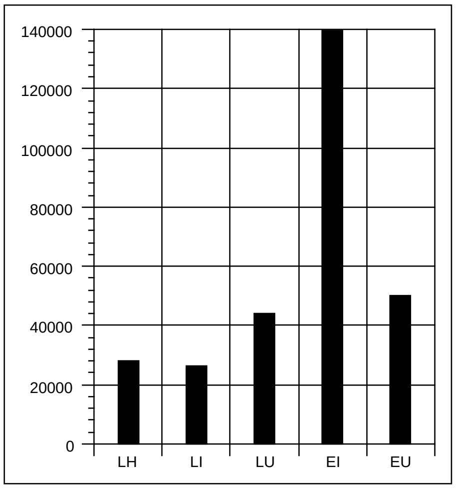


**Figure 18** Data (Kbytes) Transmitted in Cholesky

invalidate performs better than update. EU sends an order of magnitude more messages than any of the other protocols because releases cause updates to be sent to many other processors. Ninety-one percent of EU's messages are updates sent during lock releases. The invalidate protocols send fewer messages because fewer processors cache each page.

Cholesky is a program with fine-grained synchronization that uses a task queue approach to parallelism. Locks are used to dequeue tasks as well as to protect access to multiple columns of data. Figures 16-18 summarize Cholesky's performance. The large amount of synchronization limits the speedup to no more than 1.3 for any of the protocols. The eager protocols suffer from excessive updates and invalidations caused by false sharing. The lazy protocols, and in particular LH, fare better because communication is largely localized to the synchronizing processors, leading to much better handling of false sharing.

Our simulations indicate that synchronization is a major obstacle to achieving good performance on DSM systems. For example, 83% of the messages required by Water running on the 16-processor *ATM* model under the hybrid protocol were for synchronization. For Cholesky running on 2 processors, 96% of the messages were used for synchronization. All but a few of these synchronization messages were for lock acquisition. Moreover, 84% of each processor's time was spent acquiring locks in the 16-processor LH Cholesky run. While approximately one third of the lock acquisition messages carried data, the rest were solely for synchronization purposes. When a lock is reacquired by the same processor before another processor acquires it, the lazy protocols have an advantage over the eager protocols. An eager protocol must distribute diffs at every lock release. Lazy release consistency permits us to avoid external communication when the same lock is reacquired.

### 6.3 The Effect of Network Characteristics

The network is a shared resource that can be a performance bottleneck. We can break down the network's effect on performance into three categories: *bandwidth*, *serialization*, and *collisions*. Bandwidth affects the total amount of data that can be moved. Serialization refers to the processor wait time when other processors have control of the contended network link. By *collisions* we mean actual network collisions as well as the effect of protocols like exponential backoff that are used to avoid network collisions in the case of an ethernet network. Table 2 summarizes speedup for Jacobi and Water on five different networks.

| | Jacobi | Water |
|---|---|---|
| 10 Mbit Ethernet w/ Coll. | 2.5 | 0.7 |
| 10 Mbit Ethernet w/o Coll. | 4.1 | 1.3 |
| 10 Mbit ATM | 10.1 | 4.0 |
| 100 Mbit ATM | 13.7 | 8.3 |
| 1 GBit ATM | 13.8 | 8.8 |

**Table 2** Speedups With Different Network Characteristics (LH, 16 processors)

Jacobi communicates with neighbors at a barrier. Both the implementation of barriers and the access pattern (regular, to fixed neighbors) benefit from a point-to-point network that eliminates most serialization. Hence, most of the benefits of ATM for this program are from the concurrency in the network. Water's access pattern is much less regular because molecules move. The potential for communication to be completed entirely in parallel is significantly reduced. As a result, Water benefits as much from network concurrency as from increased bandwidth. Increasing the network bandwidth to 1 Gbit/sec does not improve performance significantly with a 40 MHz processor, since at this point, the software overhead is the major performance bottleneck.

## 6.4 The Effect of Software Overheads

Software overheads have a significant impact on performance. Table 3 shows the simulated performance of an ATM network in the 16-processor case, with no software overhead, with software overhead identical to that used in the previous simulations, and with double that amount.

We first removed the overhead in order to find an upper bound on DSM performance for the given network and processor architecture, regardless of the operating system and DSM implementation. The large speedups indicate the performance potential for the protocols, and the potential gains to be had from hardware support.

With software overhead removed, there is no longer a significant per-message penalty on a crossbar network. This lessens the importance of access misses, and favors protocols that reduce the amount of data moved for improved performance. For instance, the LI protocol outperforms LH on a 16-processor Cholesky run even though the LH protocol sends 30% fewer messages and has 75% fewer access misses than the LI protocol. The reason is that the hybrid protocol attempts to find a compromise between low message counts, low numbers of access misses, and low amounts of data, but the data total is more significant if software overhead is removed.

| Prog. | Overhd. | LH | LI | LU | EI | EU |
|---|---|---|---|---|---|---|
| Jacobi | Zero | 15.1 | 15.3 | 15.1 | 14.9 | 15.4 |
| | Normal | 13.7 | 13.4 | 13.7 | 14.2 | 13.4 |
| | Double | 12.9 | 12.6 | 12.8 | 12.7 | 12.5 |
| TSP | Zero | 7.8 | 7.8 | 7.8 | 10.3 | 10.3 |
| | Normal | 7.9 | 7.9 | 7.8 | 9.7 | 9.8 |
| | Double | 8.7 | 8.7 | 7.4 | 10.3 | 10.3 |
| Water | Zero | 13.1 | 13.1 | 12.8 | 5.2 | 10.5 |
| | Normal | 8.3 | 7.7 | 6.6 | 3.8 | 2.6 |
| | Double | 6.9 | 6.0 | 5.2 | 3.3 | 1.5 |
| Chol. | Zero | 2.4 | 2.6 | 1.2 | 0.7 | 1.3 |
| | Normal | 0.8 | 0.7 | 0.4 | 0.5 | 0.2 |
| | Double | 0.4 | 0.4 | 0.2 | 0.3 | 0.1 |

**Table 3** Speedups With Varying Software Overhead (16 processors)

The significance of software overhead can be seen most clearly in comparing the speedups of Water with and without overhead. The lazy protocols improve by an average of 80% when the overhead is removed. EI still performs badly because the amount of data it moves, five times more than any of the other protocols. EU, which runs three times slower than the LH protocol when software overhead is included, speeds up by more than 400% when software overhead is removed.

In order to determine the variation in performance that might occur due to an increase in software overhead, we determined speedups when the overhead per message was doubled. The performance decreases by 20% to 40% for Water. The decrease in performance is not as large as when going from zero to normal overhead since the normal overhead includes the per diff overhead, which is significant. In general, the lazy protocols, and in particular the lazy hybrid, perform better as communication becomes more expensive.

## 6.5 The Effect of Processor Speeds

Processor speeds affect the ratio of computation time to communication time. However, the software overhead is proportional to the processor speed. We varied the processor speeds from 20 to 80 MHz. Table 4 shows the variation in speedup for the 16-processor case when using the lazy hybrid protocol in the case of Jacobi, TSP and Water, and the 8-processor case for Cholesky. For Jacobi and TSP, the variations are negligible because the low message counts for these programs results in little variation in the computation to communication ratio. Water and Cholesky show a more significant variation in speedup due to the larger amount of communication. In the latter two cases, communication latency is as much of a bottleneck as the software overheads, and hence an increased processor speed reduces speedup. However, some of the improvements are masked by the corresponding changes in software overheads.

### 6.6 The Effect of Page Size

The large page sizes in common use in software DSMs result in a high probability of false sharing. Prior work has developed implementations of relaxed memory consistency models for DSM that reduce but do not totally eliminate the effects of false sharing. For example, Munin's eager implementation of release consistency eliminates the "ping-pong" effect of a page bouncing between two writing processors [5]. However, modifications to falsely shared pages still have to be distributed to all processors caching the page at a release. The lazy hybrid protocol further reduces the effect of false sharing because data movement only occurs between synchronizing processors. In other words, false sharing in LH increases the amount of data movement but *not* the number of messages.

The results we have reported are for a page size of 4096 bytes. To obtain a measure of the effects of false sharing, we ran simulations using a page size of 1024 bytes. While going to a 1024-byte page reduces false sharing, we found that we need to communicate with approximately the same number of processors to maintain consistency. Furthermore, the resulting reduction in communication is often partially counterbalanced by the increased number of access misses (see Table 5, which presents data for the lazy hybrid protocol). While reducing the page size has a limited effect on performance, restructuring the program may prove more beneficial.

| Procs | Page Size (bytes) | Jac. | TSP | Wat. | Chol. |
|---|---|---|---|---|---|
| 2 | 1024 | 1.8 | 1.7 | 1.9 | 1.0 |
| | 4096 | 1.8 | 1.7 | 1.8 | 0.9 |
| 4 | 1024 | 3.7 | 2.6 | 3.1 | 1.2 |
| | 4096 | 3.7 | 2.6 | 2.9 | 1.1 |
| 8 | 1024 | 7.2 | 5.1 | 5.1 | 1.4 |
| | 4096 | 7.2 | 5.1 | 4.8 | 1.3 |
| 16 | 1024 | 13.7 | 8.5 | 8.7 | 0.9 |
| | 4096 | 13.7 | 7.9 | 8.3 | 0.8 |

**Table 5** Effect on Speedup of Reducing the Page Size to 1024 bytes (LH)

## 7 Related Work

This work draws on the large body of research in relaxed memory consistency models (e.g., [2, 4, 8, 9]). We have chosen as our basic model the release consistency model introduced by the DASH project at Stanford [12], because it requires little or no change to existing shared memory programs. An interesting alternative is *entry consistency* (EC), defined by Bershad and Zekauskas [4]. EC differs from RC because it requires all shared data to be explicitly associated with some synchronization variable. On a lock acquisition EC only needs to propagate the shared data associated with the 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.

Ivy [13] and Munin [5] are two implementations of software DSMs for which performance measurements have been published. Both achieve good speedups on many of the applications studied. The slow processors used in the implementations prevented the network from becoming a bottleneck in achieving these speedups. With faster processors, faster networks are needed and more sophisticated methods are required. In addition, synchronization latency becomes a major issue. Performance measurements are also available for the DASH hardware DSM multiprocessor. Comparison between these numbers and our simulation results indicates the benefits of a dedicated high-speed interconnect for fine-grained parallel applications.

| Pr. Spd (MHz) | Jacobi | TSP | Water | Chol. |
|---|---|---|---|---|
| 80 | 13.7 | 10.5 | 7.7 | 0.9 |
| 40 | 13.7 | 9.8 | 8.3 | 1.3 |
| 20 | 13.4 | 10.0 | 8.6 | 1.4 |

**Table 4** Speedups with Different Processor Speeds (LH, 16 processors)

## 8 Conclusions

With the advent of faster processors, the performance of DSM that can be achieved on an Ethernet network is limited. Serialization of messages, collisions, and low bandwidth severely constrain speedups, even for coarse-grained problems. Higher-bandwidth point-to-point networks, such as the ATM LANs appearing on the market, allow much better performance, with good speedups even for medium-grained applications. Fine-grained applications still perform poorly even on such networks because of the frequency and cost of synchronization operations.

*Lazy hybrid* is a new consistency protocol that combines the benefits of invalidate protocols (relatively little data) and update protocols (fewer access misses and fewer messages). In addition, the lazy hybrid shortens the lock acquisition latency considerably compared to a lazy update protocol. The hybrid protocol outperforms the other lazy protocols under a model that takes into account software overhead for communication. For medium-grained applications the differences are quite significant.

The latency of synchronization remains a major problem for software DSMs. Without resorting to broadcast, it appears impossible to reduce the number of messages required for lock acquisition. Therefore, the only possible approach may be to hide the latency of lock acquisition. Multithreading is a common technique for masking the latency of expensive operations, but the attendant increase in communication could prove prohibitive in software DSMs. Program restructuring to reduce the amount of synchronization may be a more viable approach.

# References


[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. Technical Report CS-1051, University of Wisconsin, Madison, September 1991.

[2] M. Ahamad, P.W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281, May 1991.

[3] H.E. Bal and A.S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[4] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[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] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[7] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The Efficient Simulation of Parallel Computer Systems. *International Journal in Computer Simulation*, 1:31–58, January 1991.

[8] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Computers*, 16(6):660–673, June 1990.

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

[10] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software: Practice and Experience*, 23(2):201–221, February 1993.

[11] 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.

[12] 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.

[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] 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.