RICE UNIVERSITY

# Lazy Release Consistency for Distributed Shared Memory

by

**Peter Keleher**

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:

Willy Zwaenepoel,
Professor, Chairman
Computer Science

Alan Cox, Assistant Professor
Computer Science

John Bennett, Associate Professor
Electrical and Computer Engineering

Houston, Texas

January, 1995

# Lazy Release Consistency for Distributed Shared Memory

Peter Keleher


## Abstract

A software distributed shared memory (DSM) system allows shared memory parallel programs to execute on networks of workstations. This thesis presents a new class of protocols that has lower communication requirements than previous DSM protocols, and can consequently achieve higher performance. The lazy release consistent protocols achieve this reduction in communication by piggybacking consistency information on top of existing synchronization transfers. Some of the protocols also improve performance by *speculatively* moving data.

We evaluate the impact of these features by comparing the performance of a software DSM using lazy protocols with that of a DSM using previous *eager* protocols. We found that seven of our eight applications performed better on the lazy system, and four of the applications showed performance speedups of at least 18%. As part of this comparison, we show that the cost of executing the slightly more complex code of the lazy protocols is far less important than the reduction in communication requirements. We also compare the lazy performance with that of a hardware supported shared memory system that uses processors and caches similar to those of the workstations running our DSM. Our DSM system was able to approach, and in one case even surpass, the performance of the hardware system for applications with coarse-grained parallelism, but the hardware system performed significantly better for programs with fine-grained parallelism.

Overall, the results indicate that DSMs using lazy protocols have become a viable alternative for high-performance parallel processing.

## Acknowledgments

I wish to thank my committee, Willy Zwaenepoel, Alan Cox, and John Bennett, for their guidance on my thesis. Together with Sandhya Dwarkadas, they have provided me with advice, encouragement, and support throughout my graduate career. I am also indebted to them for the opportunity of working in a first-class research environment, whose excellence I am starting to appreciate only as I leave Rice.

My time at Rice was made special by many friends. Ervan, John, Nat, Mootaz, and Uli all helped make life as a graduate student a Good Thing. However, I may not have survived without Chau-Wen, who showed the way out, Reinhard, who taught that a hike in the mountains or a good bike ride cures all ills, and Jerry, who was always happy to match his frustrations against mine on the squash court.

Most of all, I would like to thank my wife, who makes everything worthwhile.

# Contents

# Illustrations

# Tables

# Chapter 1

# Introduction

The current trend in the supercomputer industry is to build parallel machines using off-the-shelf processors and a high-performance interconnect. The primary reason for this is that off-the-shelf microprocessors are now nearly as fast as the best custom chip sets. A relatively small number of these microprocessors linked together by a high-performance interconnect can provide a shared address space and very high aggregate performance. Moreover, using commonly available microprocessors provides a clear upgrade path as new processors are produced.

As a result of these trends, the same processors used in supercomputers are often used in ordinary workstations on local area networks. Supporting the abstraction of shared memory on these high-performance *multicomputers* is an obvious idea. However, local area networks have historically lacked the high bandwidth and low latency of supercomputer interconnects, as well as the hardware support for implementing shared memory over the message-passing interconnect. Conventional wisdom therefore says that distributed shared memory (DSM) systems, which provide shared address spaces to networks of workstations (see Figure 1.1), can not be implemented efficiently.

This thesis develops a family of *lazy release consistent* (LRC) protocols that effectively address many of the problems inherent in supporting DSM on a network. LRC protocols delay communicating consistency information until absolutely required to do so by the memory model. At the other end of the spectrum are *eager* protocols, which attempt to minimize latency by optimistically moving data before it is actually needed. By moving data and consistency information only upon request, the lazy protocols often require significantly less communication than eager protocols. The primary drawback of this approach is that hiding communication latency by overlapping communication with computation becomes more difficult. However, we have found that for many classes of programs, the reduction in message traffic achieved by the lazy protocols more than offsets any added latency.

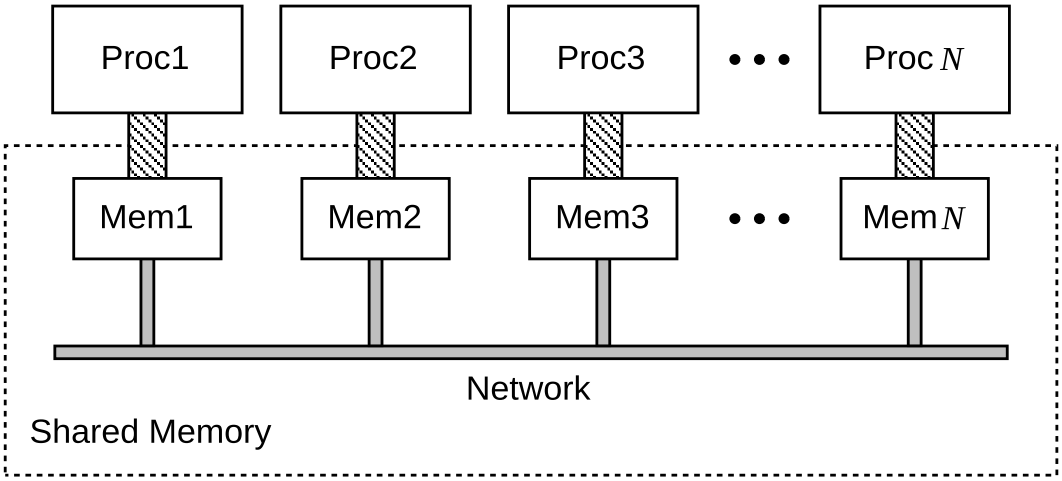


**Figure 1.1** Distributed Shared Memory

The LRC protocols have been implemented on a network of eight workstations, and timing information from the implementations has been used to calibrate simulations of up to 64-processor clusters of workstations. Our results show that efficiently supporting the abstraction of shared memory in software is possible for a broad class of applications, but only with the use of high-performance protocols that are specifically crafted to reduce communication requirements. Given appropriate protocols, DSM systems can have performance comparable to hardware systems [LT88] for small clusters, and significant speedup even on large clusters.

## 1.1 Programming Model

The LRC protocols developed in this work provide a generic abstraction of shared memory to application programs. Programs are multi-threaded, and synchronize through locks and barriers. All data communication between threads is through globally shared memory. For most programs, the memory abstraction supported by LRC protocols is indistinguishable from that of a multiprocessor that supports shared memory in hardware [LT88].

The protocols and the system described in this dissertation do not require user annotation or language support. For several reasons, we feel that it is important for DSM systems to transparently run programs written for hardware shared memory systems.

First, the overriding rationale for using DSM systems is that they are easier to use than message-passing systems. DSMs handle data movement automatically, while message-passing systems require data movement to be specified by the programmer. Requiring users to annotate shared variables or sharing patterns may negate the underlying rationale of DSMs.

Second, we believe that low-level customization hooks can and should allow experienced users to tune system performance. However, requiring all programmers to reason about the underlying implementation would likely be counterproductive.

Finally, none of the work described in this thesis disallows later addition of application-specific semantic information. In fact, most such optimizations are orthogonal to the decisions made in this thesis. Beginning without any application-specific information forces us to reason clearly about general application characteristics, and to identify and address common sharing patterns. Once general-purpose mechanisms that efficiently support large classes of applications are in place, special-purpose mechanisms that exploit application-specific information can be added to bring the performance as close to optimal as possible.

## 1.2 Challenges

The following paragraphs describe the two primary obstacles to obtaining good performance on a software implementation of distributed shared memory.

### Communication

The high cost of communication on local area networks can hurt performance in two ways. First, large amounts of communication can cause the network to become a bottleneck. Historically, network multicomputers have had less bandwidth and higher latency than hardware shared memory (HSM) machines. However, the ATM networks used by our implementation and modeled in the simulations are fast enough that the actual *wire time* of messages is always dwarfed by operating system costs.

More important on our target systems is the *software overhead* incurred any time a message is sent or received. Each message send, for example, requires traversing many different levels of system software from the kernel trap down to the network interface, often making the actual wire time an insignificant contributor to the overall transmission cost. Lazy protocols are ideally suited for situations where communication has a high per message cost, because they send messages only when absolutely necessary, often resulting in far fewer messages overall.

**False Sharing**

False sharing results when the system can not distinguish between accesses to logically distinct pieces of data. False sharing occurs because the system tracks accesses at a granularity larger than the size of individual shared data items. Conventional protocols typically require processes to gain sole access to a page before it can be modified. Therefore, false sharing can lead to situations where multiple processes contest ownership of a page, even though the processes are modifying entirely disjoint sets of data. The page may then "ping-pong" back and forth between the processes.

The LRC protocols presented in this thesis allow processes to modify pages without gaining sole ownership of a page. Multiple processes can thereby modify falsely shared data simultaneously, without network communication. False sharing is more common on DSM systems than on HSM systems because DSMs track accesses at the granularity of virtual memory pages, while HSMs track accesses at the granularity of cache lines.

## 1.3 Lazy Release Consistency

Lazy release consistency is based on *release consistency* (RC) [GLL+90], a relaxed memory consistency model that permits a processor to delay making its changes to shared data visible to other processors until subsequent synchronization accesses occur. Essentially, all shared accesses are divided into *ordinary* accesses, *acquire* synchronization accesses, and *release* synchronization accesses. Release consistency allows the results of ordinary shared writes to be buffered locally until the next release operation.

In contrast, *sequential consistency* (SC) [Lam79], until recently the model implemented by most bus-based multiprocessors, requires all prior shared writes to complete before any subsequent shared accesses can be initiated. RC systems can achieve large performance gains over SC systems because they allow updates to be buffered.

The primary disadvantage of using RC is that the memory abstraction seen by the user is slightly different than with SC. However, programs written for SC produce the same results on an RC memory, provided that (i) all synchronization operations use system-visible primitives, and (ii) there is a chain of synchronization between every pair of conflicting ordinary accesses to the same memory location by different processors [GLL+90]. In practice, most shared memory programs require little or no modification to meet these requirements.

LRC is a refinement of RC that allows consistency action to be postponed until a synchronization variable released in a subsequent operation is acquired by another processor. Even then, the shared writes are made visible only to the acquiring processor. Synchronization transfers in an LRC system, therefore, involve only the synchronizing processors. A release in an eager RC system requires the releasing processor to make its shared writes visible to all other processors in the system that cache the data. This reduction in synchronization traffic can result in a significant decrease in the total amount of system communication, and a consequent increase in overall performance.

## 1.4 Thesis

This dissertation centers around the following three claims:

> **Claim 1.1** DSM systems based on LRC require less communication than systems based on comparable eager protocols.

Earlier work has shown that eager release consistent (ERC) systems outperform conventional DSM systems and can approach explicit message passing in communication requirements [Car93]. We therefore use ERC protocols as the yardstick with which to gauge the success of the LRC protocols.

LRC protocols piggyback consistency information on top of synchronization messages, and only move data when needed. Eager protocols attempt to hide communication latency by moving data ahead of any need. Our work shows that for a broad class of systems, the extra communication needed to move data eagerly overshadows any gains made in hiding latency.

> **Claim 1.2** DSM systems based on LRC can achieve better performance on a broader range of applications than systems based on eager protocols.

Many previous systems either restricted their focus to coarse-grained programs [LH89, FP89], required user annotations [ZSB94, Lee94], or substantially changed the programming model [BT88, DCM+90]. Each of these choices has merit, but none of them is ideal from the standpoint of transparently handling a broad range of programs.

LRC protocols can efficiently handle complicated sharing patterns and data layouts without user annotations or changes in the programming model because they reduce the communication impact of synchronization operations. Hence, LRC allows a broader range of existing programs to be run efficiently.

> **Claim 1.3** DSM systems can often obtain performance competitive with hardware shared memory systems.

Claim 1.3 has been made before, but this work supersedes previous work in that (i) we show that the lazy protocols usually perform substantially better than the eager protocol [DKCZ93, Car93] that represents the previous state of the art, and (ii) we back up our contention by presenting a detailed performance comparison of our system with a hardware shared memory system that is based on the same processors and caches, as well as extensive simulations.

## 1.5 Contributions

The primary contributions of this dissertation are the design, implementation, and evaluation of the LRC protocols, and the consequent validation of the claims made in Section 1.4.

Claim 1.1 says LRC systems require less communication than ERC systems. In order to validate this contention, we extensively simulated several variants of each type of system, and then built TreadMarks and analyzed its performance. TreadMarks is an LRC-based DSM system that runs on standard workstations and operating systems.

We also built eager versions of TreadMarks, and analyzed differences in performance and communication behavior between the variants. Our results show that there is little difference between lazy and eager systems for coarse-grained programs, but the difference grows substantially as the programs become more fine-grained, either in synchronization or in data sharing. Seven of the eight applications in our suite performed better on the LRC system than on the eager, and four of those programs performed at least 18% better. Although there are certainly classes of programs for which eager protocols would require less communication than LRC protocols, none of our programs displayed this behavior.

Claim 1.2 contends that LRC protocols can efficiently execute a broader range of programs than eager systems. Our application suite includes a wide variety of programs drawn from several different sources. The programs range from fine-grained lock-based programs to coarse-grained programs that use only barriers. The results of Chapter 3 show that, with one exception, the LRC protocols consistently outperform the eager protocols over all combinations of sharing and synchronization patterns.

Finally, Claim 1.3 makes the controversial contention that DSM systems can perform comparably to hardware-based systems. Our defense of this claim rests on two studies.

The first is a comparison of the performance of TreadMarks to that of a SGI shared memory machine that uses the same processors, primary caches, and compiler as the machines running TreadMarks. The primary difference between the two systems is the way the shared memory abstraction was implemented. Our results show that TreadMarks performs better than the SGI for one program, and nearly as well for several others. However, the SGI performs substantially better for the most fine-grained programs.

The second experiment was a simulation study driven by numbers from the implementations. We were able to confirm our conclusions from the performance study and pinpoint where such factors as network bandwidth become a limiting factor in achieving good performance for software systems.

Additionally, we investigated hybrid systems that used hardware shared memory in small-scale clusters, and software DSM in between clusters. Our results show that software system performance drops off rapidly as system size increases. However, hybrid systems can run coarse and medium-grained programs nearly as fast as hardware for up to 64 processes. The fine-grained programs performed little better on the hybrid system than on the software system, primarily because synchronization accesses usually require network communication in either case.

# Chapter 2

# Lazy Release Consistency

We describe the LRC memory model and present a qualitative argument arguing that protocols implementing LRC require less communication, and hence achieve better performance, than protocols implementing other memory models. We then present the design and implementation of three software DSM protocols: lazy invalidate (LI), lazy hybrid (LH), and eager invalidate (EI). LI and LH are new protocols that implement LRC. EI is a straightforward, invalidate-based implementation of eager release consistency (See Section 2.3.3). EI is used as the basis for our comparisons in Chapter 3 because studies [CBZ91, Car93] have shown that eager update protocols uniformly perform better than conventional protocols, and our own work [KCZ92, DKCZ93] has shown that eager invalidate protocols outperform eager update protocols.

Section 2.1 describes the user interface of TreadMarks, the DSM system in which all three protocols are implemented. Section 2.3 describes the design and implementation of the three protocols. Section 2.4 presents a proof that LRC is indistinguishable from conventional memory models under most conditions, and Section 2.5 summarizes the chapter.

## 2.1 Application Program Interface (API)

TreadMarks is entirely implemented as a C library, using an interface similar to the `parmacs` macros from Argonne National Laboratory [Lea87] for process and synchronization support. While the `parmacs` macros are implemented using `m4` macros, our DSM library is implemented as a set of procedure calls. Nonetheless, properly synchronized programs using the `parmacs` macros can be ported to our system with only minor changes in naming and initialization.

TreadMarks programs follow a conventional shared memory style, using processes to express parallelism, and locks and barriers to synchronize. Typically, the *manager* process initializes the DSM system, allocates and initializes shared memory, and then starts a single remote process on each remote processor via `Tmk_startup`.

The manager process passes the address of the shared region to other processes through a `Tmk_distribute` call. After initialization is complete, the parent and children each perform a portion of the work, communicating only through synchronization operations (`Tmk_lock_acquire` and `Tmk_lock_release`, `Tmk_barrier`).

The Figure 2.1 shows a complete, runnable program that fills an array in parallel.

## 2.2 Lazy Release Consistency

### 2.2.1 Motivation and Background

Most previous DSM systems supported the canonical consistency model, *sequential consistency* (SC) [Lam79]. However, many studies have shown that sequential consistency poses serious problems for efficient distributed implementations of shared memory, primarily because sequential consistency imposes such strict requirements on system-wide ordering of accesses to shared memory [Adv93]. Sequential consistency is defined as follows:

> **Definition 2.1 Sequential Consistency**
> A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

This definition can be paraphrased as requiring all shared accesses to be consistent with some total ordering, such that this total ordering does not violate program order. SC does not require an implementation to actually establish a total ordering on all accesses, but it does require all reads to return values that are consistent with such an ordering. For instance, Figure 2.2 shows an execution that violates SC, assuming each operation occurs atomically and program order is respected. $P_1$ and $P_2$ disagree on the ordering of the writes. $P_1$ can only conclude that $w_1 \prec r_1 \prec w_2 \prec r_2$, while $P_2$ concludes $w_2 \prec r_2 \prec w_1 \prec r_1$. Figure 2.3, on the other hand, is a valid sequentially consistent result because both processes can agree on an ordering that is consistent with the returned values.

```
/*
 * File app.c
 */
#include "Tmk.h"

extern  char   *optarg;

int     arrayDim = 100;
int    *array;

void main(int argc, char **argv)
{
    int     c, start, end, i;

    while ((c = getopt(argc, argv, "d:")) != -1)
        switch (c) {
        case 'd':
            arrayDim = atoi(optarg);
            break;
        }
    Tmk_startup(argc, argv);

    if (Tmk_proc_id == 0) {
        array = (int *) Tmk_malloc(arrayDim * sizeof(int));
        Tmk_distribute(&array, sizeof(array)); /* Send 4-byte ptr value */
    }
    Tmk_barrier(0);

    start = Tmk_proc_id * (arrayDim / Tmk_nprocs);
    end = (Tmk_proc_id + 1) * (arrayDim / Tmk_nprocs);
    if (end > arrayDim) end = arrayDim;

    for (i = start; i < end; i++)
        array[i] = i;

    Tmk_barrier(0);
    Tmk_exit(0);
}
```

**Figure 2.1** Simple DSM program

| $\mathbf{P_1}$ | $\mathbf{P_2}$ |
|---|---|
| $w_1(x)2$ | $w_2(y)2$ |
| $r_1(y)0$ | $r_2(x)0$ |

**Figure 2.2** Not SC ($x, y$ initially zero)

| $\mathbf{P_1}$ | $\mathbf{P_2}$ |
|---|---|
| $w_1(x)2$ | $w_2(y)2$ |
| $r_1(y)0$ | $r_2(x)2$ |

**Figure 2.3** SC ($x, y$ initially zero)

While useful as a description of a base memory model, sequential consistency is no longer commonly used in new parallel machines. Many hardware optimizations that are used to hide memory access latency violate SC. For instance, the example in Figure 2.2 could be produced by a machine that has write buffers, and allows reads to bypass writes in the buffers.

Supporting SC in the presence of non-atomic memory transactions is even more difficult. An update in a distributed system can be logically decomposed into a series of sub-operations [Col91], each of which applies to a single process. If the relative orderings of sub-operations of competing memory operations do not agree, then SC is violated. This problem is especially severe in systems in which sub-operations take differing amounts of time to complete, such as ring architectures that support concurrent access [WHL92], or NUMA machines [BSF+91, SJG92, BFS89, Cox92, CF89, LE91, LEK91] .

### 2.2.2 Release Consistency

Release consistency (RC) [GLL+90], hereafter referred to as *eager* release consistency, is a *relaxed* memory consistency model that permits a process to delay making its changes to shared data visible to other processes until certain synchronization accesses occur. At an intuitive level, RC allows views of shared memory by different processes to become inconsistent until subsequent synchronization events.

A useful memory model must effectively address ease of use as well as performance. Relaxed consistency models are attractive because they allow better performance than previous models. However, if the resulting programming model is non-intuitive, the lack of programming effectiveness offsets any gains in performance.

Eager release consistency addresses the performance issue by allowing write accesses to be pipelined or batched, and addresses the programming model issue by guaranteeing results equivalent to an SC system for *properly-labeled* [GLL+90] programs. Informally, a program is properly-labeled if the program contains enough synchronization to avoid data races. This concept is similar to the notion of *data-race-free*, which will be discussed in Section 2.2.4.

Figure 2.4 shows how RC categorizes shared memory accesses. In order to explain this categorization, we first explain the notion of *competing accesses*. Two shared accesses by different processes *compete* if they apply to the same location and at least one is a `write`. `Sync` accesses are competing accesses used to enforce ordering or atomicity among multiple processes. `Nsync` accesses are competing accesses that do not enforce orderings, such as competing accesses to neighbor data in chaotic relaxation algorithms. `Sync` accesses are further divided into `acquires` and `releases`, `acquires` being used to gain access to shared data, and `releases` being used to grant such accesses. Ordinary accesses are those that do not compete in executions of properly labeled programs.

**Definition 2.2 Conditions for Eager Release Consistency**
(A) Before an ordinary `read` or `write` access is allowed to perform with respect to any other process, all previous `acquire` accesses must be performed, and
(B) before a `release` access is allowed to perform with respect to any

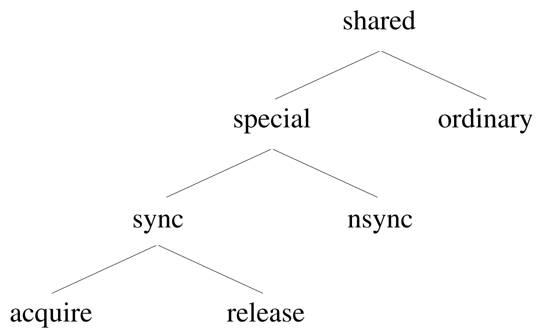


**Figure 2.4** Categorization of Writable Accesses

other process, all previous ordinary `read` and `store` accesses must be performed, and

(C) `sync` accesses are sequentially consistent with respect to one another.

Informally, a shared access is *performed* at a process when its result is visible at the process. `Acquires` and `releases` may be thought of as conventional synchronization operations on a lock, or $P's$ and $V's$ on binary semaphores, but other synchronization mechanisms can be expressed as well. For instance, the arrival of a worker process at a barrier can be modeled as a `release` by the worker followed by an `acquire` by the manager, and departure of a worker from a barrier can be modeled as a release by the manager followed by an `acquire` by the worker. Essentially, RC requires ordinary shared memory updates by a process $p$ to become visible at other processes only when a subsequent `release` by $p$ becomes visible at another process, $q$.

### 2.2.3 Lazy Release Consistency

While eager release consistency allows quite a bit of latitude in deciding when to perform ordinary shared accesses, it still requires accesses to be performed globally before a local `release` can complete. *Lazy release consistency* is a refinement of eager release consistency that allows synchronization transfers to take place without performing any ordinary shared accesses globally. Instead, the shared accesses only have to be performed at other processes as they synchronize with the performing process.

**Definition 2.3 Conditions for Lazy Release Consistency**

(A) Before an ordinary `read` or `write` access is allowed to perform with respect to another process, all previous `acquire` accesses must be performed *with respect to that other process*, and

(B) before a `release` access is allowed to perform with respect to any other process, all previous ordinary `read` and `store` accesses must be performed *with respect to that other process*, and

(C) `sync` are sequentially consistent with respect to one another.

Definition 2.2 requires ordinary accesses to be performed globally at the next release, whereas Definition 2.3 requires only that ordinary accesses be performed with respect to other processes as subsequent releases become visible to them.

Figure 2.5 shows three processes exchanging synchronization in an eager release consistent DSM using an invalidate protocol. Each process caches pages $x$ and $y$. Process $P_1$ modifies page $x$ and then releases a synchronization variable. At this point, Definition 2.2 *requires* that all previous ordinary accesses, the write to page $x$ in this case, be performed everywhere in the system. For an invalidate protocol, "performing" a write means invalidating other copies, so invalidate messages are sent to all other processes that cache $x$. When $P_2$ likewise modifies page $y$ and performs a release, invalidate messages must again be sent to all other processes that cache the affected page.

Unnecessary communication takes place at two levels in this example. First, all three invalidation messages are useless in the sense that the targets of the invalidations never access the invalidated data, and hence would not notice if their copies became inconsistent. Second, the invalidation of $x$ to $P_2$ travels the same route as a subsequent lock transfer, so a pair of messages could be eliminated by piggybacking the invalidation to the synchronization transfer.

Figure 2.6 shows the same example under an LRC invalidate protocol. Consistency management is moved from releases to subsequent acquires, and invalidates are sent only to the acquiring process. Even more importantly, LRC is able to combine the invalidation and the synchronization transfer into a single message pair because they occur at the same time.

This simple example shows some of LRC's considerable potential to reduce communication requirements over that needed by ERC. Eager systems flush modifications globally prior to releases, but lazy systems only pass consistency information between

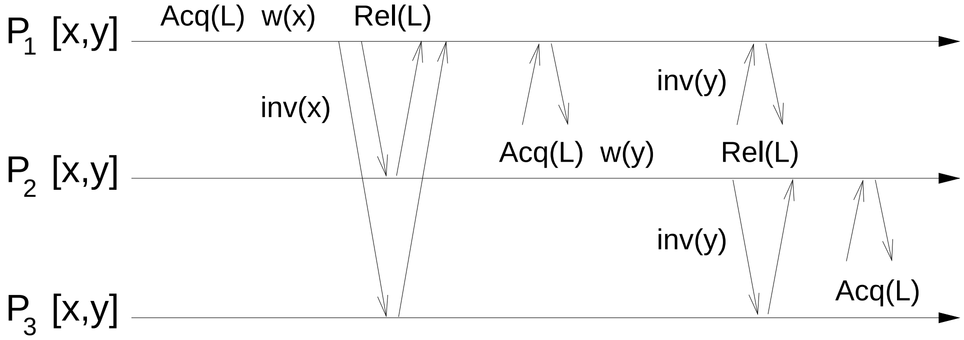


**Figure 2.5** Eager Release Consistency

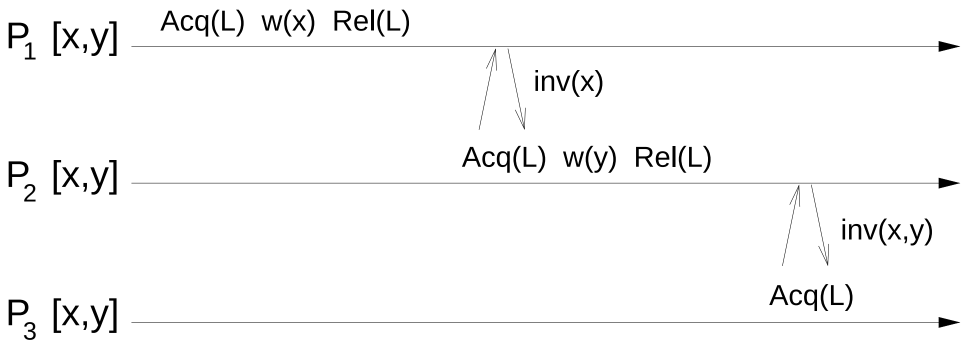


**Figure 2.6** Lazy Release Consistency

processes that synchronize with each other. LRC systems therefore need to maintain transitive information. In Figure 2.6, the lock grant from $P_2$ to $P_3$ not only carries an invalidation for $y$, which was modified by $P_2$, but also an invalidation for $x$, which was previously modified by $P_1$.

### 2.2.4 Happened-Before-1

In order to support the memory model described in Definition 2.3, we use a *happened-before-1* [AH93] partial ordering over all shared accesses:

> **Definition 2.4** Shared memory accesses are partially ordered by *happened-before-1*, denoted $\xrightarrow{hb1}$, defined as follows:
>
> - If $a_1$ and $a_2$ are accesses on the same process, and $a_1$ occurs before $a_2$ in program order, then $a_1 \xrightarrow{hb1} a_2$.
> - If $a_1$ is a release on process $p_1$, and $a_2$ is an acquire on the same memory location on process $p_2$, and $a_2$ returns the value written by $a_1$, then $a_1 \xrightarrow{hb1} a_2$.
> - If $a_1 \xrightarrow{hb1} a_2$ and $a_2 \xrightarrow{hb1} a_3$, then $a_1 \xrightarrow{hb1} a_3$.

The happened-before-1 relation is the transitive closure of program order and synchronization order (i.e. an acquire is ordered after the last previous release of the same synchronization variable).

LRC requires that before a process may continue past an acquire, all shared accesses that precede the acquire according to $\xrightarrow{hb1}$ must be performed at the acquiring process, where "performing" an access at process $p$ means either updating or invalidating $p$'s copy of the indicated data item.

LRC protocols guarantee to support the same programming model as sequentially consistent protocols if programs are *data-race-free*. The following definitions are from Adve [AH93]:

> **Definition 2.5** A *data race* in an execution is a pair of conflicting operations, at least one of which is to data, that is not ordered by the happened-before-1 relation defined for the execution. An execution is *data-race-free* if and only if it does not have any data races. A program is *data-race-free* if and only if all its sequentially consistent executions are data-race-free.

Data-race-free programs produce the same results on LRC systems as they do on sequentially consistent systems. This requirement is usually not as arduous as it may seem, because many (if not most) parallel programs are data-race-free already. Some classes of algorithms, such as chaotic algorithms, can tolerate temporary inconsistencies in their shared data. Otherwise, a data race usually represents a bug.

While satisfying Definition 2.4 is by itself enough to satisfy LRC, maintaining and using such a detailed ordering on individual shared accesses would be prohibitively expensive. Instead, the lazy protocols generalize the ordering to apply to process *intervals*. Intervals are segments of time in the execution of a single process. New intervals begin each time a synchronization access is executed by the process. We define the *happened-before-1* partial order between intervals in the obvious way: an interval $i_1$ precedes an interval $i_2$ according to $\xrightarrow{\text{hb1}}$, if all accesses in $i_1$ precede all accesses in $i_2$ according to $\xrightarrow{\text{hb1}}$. An interval is said to be performed at a process if all the interval's accesses have been performed at that process.

The lazy protocols track which intervals have been performed at a process by maintaining a per process *vector timestamp* [Mat89]. A vector timestamp consists of a set of interval indices, one per process in the system. Let $\mathrm{vv}_p^i$ be the vector timestamp of process $p$ at interval $i$. The entry for process $q \neq p$, denoted $\mathrm{vv}_p^i[q]$, specifies the most recent interval of process $q$ that has been performed at process $p$. Entry $\mathrm{vv}_p^i[p]$ is equal to $i$.

Interval $\sigma_q^x$, or interval $x$ of processor $q$, is termed *covered* by $\mathrm{vv}_p^i$ if $\mathrm{vv}_p^i[q]$ is greater than or equal to $x$. We also use the notation $\epsilon$ to represent *covered*.

The lazy protocols pass consistency information in the form of *write notices* that are attached to intervals. A write notice is an indication that a given page has been modified. Each interval contains a write notice for every page that was modified during the segment of time corresponding to the interval. Write notices are used in the `send_set`, which is the set of all write notices created during intervals that have been performed at the releasing process, but not at the acquiring process. If $\mathrm{vv}_r^i$ is the vector timestamp of a release process and $\mathrm{vv}_a^j$ is the vector timestamp of the corresponding acquire process, then the `send_set` consists of all intervals $\sigma_p^x$, such that $\sigma_p^x \epsilon\, \mathrm{vv}_r^i$ and not $\sigma_p^x \epsilon\, \mathrm{vv}_a^j$. In order to create the `send_set`, vector timestamps are included on synchronization requests.

## 2.3 Protocols

A TreadMarks program consists of one or more processes communicating through sockets. In order to minimize the demands on the underlying operating system, we do not expect lightweight process support and therefore run only a single process on each machine.

At system startup, library routines create the requested number of processes on other machines, set up fully connected sockets between the processes, and register a `SIGIO` handler to handle incoming requests asynchronously. Each process allocates a large block of local memory to use as the shared mapping of the virtually shared memory. The block is located at the same address on each machine. For each page of this memory, a "manager" processor is designated.

A `SEGV` handler is registered with the operating system in order to detect and intercept write accesses to shared pages. The `SEGV` handler is called when a process tries to access data on an unmapped or protected page. The handler retrieves a valid copy of the page from the manager, adds read permission to the page, and allows the access to proceed. The `SEGV` handler is also called when a write access to a valid page is first performed. The handler creates a *twin*, or copy, of the page, and stores it in system space. A comparison of the twin and a later version of the page is used to create a *diff*, which is a run-length encoding of the differences between the two versions. The diff can then be used to update other processes' copies of the page. With the exception of the first time a processor accesses a page, a processor's version of a given page is updated exclusively by applying diffs; a new complete copy of the page is never needed.

The only strictly necessary service that a DSM needs to provide is a coherent view of shared memory. However, synchronizing through software-supported shared memory can be unreasonably slow. We therefore separate synchronization mechanisms from ordinary shared memory mechanisms. The advantages gained thereby are twofold: (i) better performance for synchronization, and (ii) the opportunity to tune shared memory consistency mechanisms to application-level synchronization (i.e. release consistency).

### 2.3.1 Lazy Invalidate

#### Pages and Data

Shared pages each have a statically assigned manager. As indicated in Figure 2.7, they have four possible states: `UNMAPPED`, `INVALID`, `READ_ONLY`, and `READ_WRITE`. At startup, all shared pages of a given processor that are not owned by that processor are in an `UNMAPPED` state.

There are three types of page faults that can occur: a *cold miss*, which occurs the first time a page is accessed by a processor that is not the manager, a *coherence miss*, which occurs when a page is accessed after it has been invalidated due to coherence actions, and a *protection fault*, which occurs when a write access occurs to a valid but `READ_ONLY` page. Pseudo-code for the `SEGV` handler, which handles all three types of faults, is shown in Figure 2.8.

On a *cold miss*, a copy of the page is retrieved from the manager and put into a `READ_ONLY` state. It does not matter whether the manager or other processors have modified the page or not.

In response to a protection fault, the `SEGV` handler changes the virtual memory page's state to `READ_WRITE`, creates a twin for the page, and saves the twin in system space (Figure 2.9). An interval structure containing a write-notice for the page is created at the next release. A write-notice is an indication that a page has been modified. A diff of the page is then created by comparing the current version of the page with the twin. The comparison resolves differences down to the granularity of a four byte word. After the diff has been created, the twin is discarded and the page is placed back into the `READ_ONLY` state.

Coherence misses occur when incoming write notices invalidate locally mapped pages. Invalidation consists of changing a page's state from `READ_ONLY` or `READ_WRITE` to `INVALID`, and removing all access rights from the page. `INVALID` pages differ from

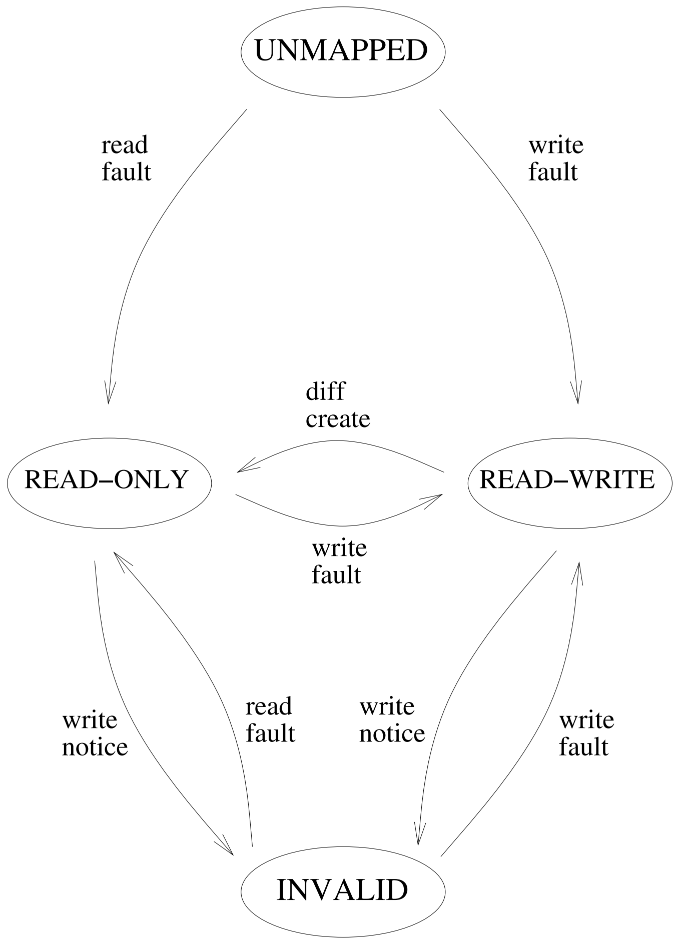


**Figure 2.7** Page State Transitions

`UNMAPPED` pages in that it is only necessary to apply a sequence of diffs to `INVALID` page to re-validate them. Valid pages are never made `UNMAPPED`.

A coherence miss indicates that at least one other processor has made changes to the page that should be reflected in the local copy before it is accessed again. In Figure 2.10, process $P_3$ takes a coherence miss on page $p$, which contains data $x$, $y$, and $z$. Unlike conventional protocols [LH89], lazy protocols allow processes to determine the location of needed data entirely on the basis of local information. $P_3$ is therefore able to determine that there have been two previous modifications that need to be applied locally before accessing $z$. Moreover, $P_1$'s modification of $x$ precedes $P_2$'s modification of $y$ via $\xrightarrow{\text{hb1}}$, so $P_2$ must have applied $P_1$'s modification of $x$ before it accessed the page to modify $y$. Since diffs are only discarded during *garbage collection*

```
if ( p READ_ONLY ) then
    Allocate twin
    page p ← READ_WRITE
else
    if ( cold miss ) then
        get copy from manager
    if ( write notices ) then
        Retrieve diffs
    if ( write miss ) then
        Allocate twin
        Change protection to READ_WRITE
    else
        Change protection to READ_ONLY
end
```

**Figure 2.8** `SEGV` handler for page $p$

(see Section 2.3.1), this information enables $P_3$ to request both diffs from $P_2$ rather than requesting the diffs from different sites. More generally, if processor $q$ modified page $p$ at interval $\sigma_q^x$, then $q$ is guaranteed to have any diffs of page $p$ created in an interval $\sigma_s^y$, such that $\sigma_s^y \xrightarrow{\text{hb1}} \sigma_q^x$. Therefore even if diffs from multiple writers need to be retrieved, it is usually only necessary to communicate with one other processor.

After the diffs have been retrieved, they are applied to the local copy of the page in an order consistent with $\xrightarrow{\text{hb1}}$ and the process is allowed to proceed.

### Locks

"Lock" and "unlock" synchronization primitives are mapped onto the acquire and release semantic notions in a straightforward manner. Each lock has a current owner, which is the last process that acquired the lock, and a manager, which tracks the current owner. Figure 2.11 shows pseudo-code for lock acquisitions. In this figure, the notation $wn_p$ denotes a write notice for page $p$.

Locks are acquired by capturing per lock *tokens*. Each process also maintains per lock fields `local` and `held` that indicate whether the token is currently owned by the process, and if so whether the lock is currently being held. If the token is local, acquiring a lock is a matter of setting a flag.

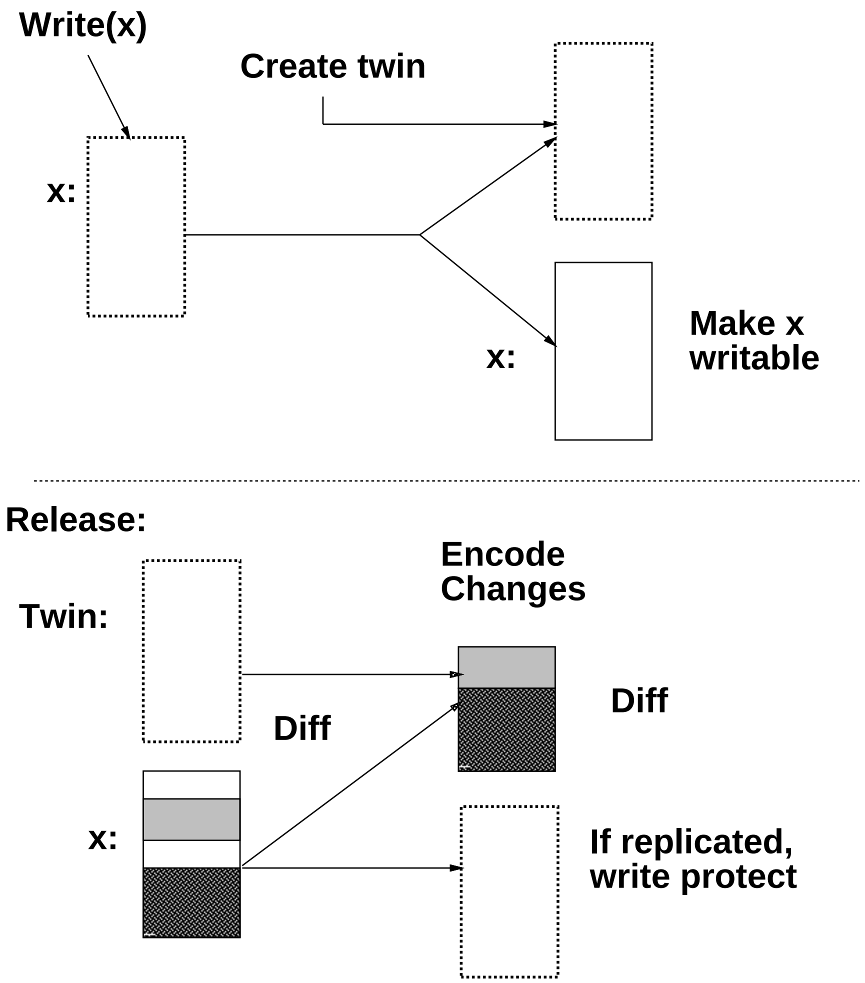


**Figure 2.9** Diff Creation

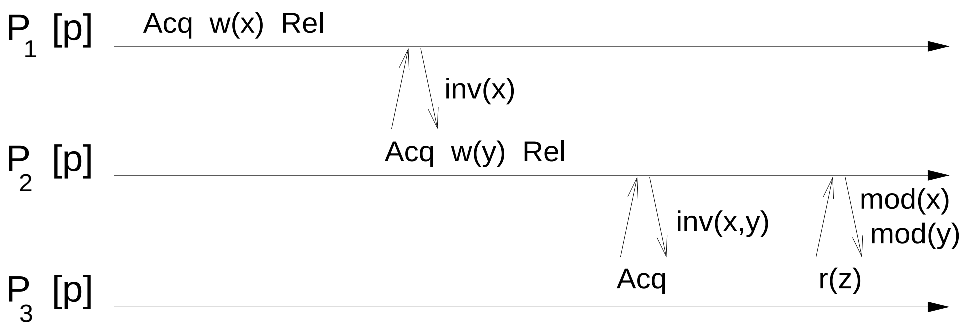


**Figure 2.10** Combining diff requests ($x$,$y$,$z$ all on page $p$)

```
if ( local ) then
    held ← TRUE
    return
end
Send request, with VV_a, to manager
Create new interval
Wait for grant
foreach interval σ_q^x ϵ send_set
    foreach wn_p ϵ σ_q^x
        if page p READ_ONLY then
            Change page p protection to INVALID
        elseif page p READ_WRITE
            remove from dirty_list
            create write notice, if necessary
            create diff for page
            de-allocate twin
            Change page p protection to INVALID
        end
    end
end
held ← local ← TRUE
```

**Figure 2.11** Lock Acquisition

If the token is non-local, a request is sent to the lock's manager, which forwards the request to the last requester of the lock. Each node maintains a `next` field per lock, and in combination, the `next` fields of the nodes waiting for a given lock implement a distributed FIFO queue of waiting processors. The acquirer creates a new interval after sending the request in order to avoid interference between write notices returning from the releaser and local pages that are in a modified state. Interval creation entails adding write notices for each dirty page to a newly created interval structure. In the general case, remote lock acquisitions take three messages, but only two are needed if the manager owns the token.

An alternative to this static ownership scheme is to use an adaptive scheme that relies on guesses of the token's location, and follows successive guesses to the current owner of the token [Car93, LH89]. Since local guesses are updated to point to the requester as the request is forwarded along, every node in the chain of guesses ends up knowing which process currently owns the token, and subsequent request chains are likely to be short. However, our experience is that this type of scheme still uses more messages than the simpler manager scheme.

Figure 2.12 shows pseudo-code for the lock request handler. The request is immediately granted if the lock is not being held and the lock's token is local to the handler's processor. Granting a lock involves creating a new local interval, determining the `send_set`, and sending the `send_set` along with the lock grant to the requester. The request is forwarded to the last processor that had previously requested the lock if it is not local, or is already spoken for.

Figure 2.13 shows the pseudo-code for a lock release. If the lock has been requested by another processor, a new interval is created, and the `send_set` and lock grant are sent to the requester.

**if** ( *local* ) **then**
    **if** ( *held* ) **then**
        save $\mathrm{vv}_a$
    **else**
        Create new interval
        *send_set* $\leftarrow$ all intervals $\sigma_q^x \in \mathrm{vv}_r$ and not $\sigma_q^x \in \mathrm{vv}_a$
        Send *send_set* and lock grant to *requester*
    **end**
**else**
    Forward request to last requester
**end**

**Figure 2.12** Lock Request Handler

Create new interval
*held* $\leftarrow$ FALSE
**if** ( there has been a request ) **then**
    *send_set* $\leftarrow$ all intervals $\sigma_q^x \in \mathrm{vv}_r$ and not $\sigma_q^x \in \mathrm{vv}_a$
    Send *send_set* and lock grant to *requester*
    *local* $\leftarrow$ FALSE
**end**

**Figure 2.13** Lock Release

### Barriers

Barriers are implemented with a centralized *barrier manager* that collects arrival messages and distributes departure messages. Managers are statically assigned to barriers in round robin order.

In terms of consistency information, a worker's barrier arrival is modeled as a release by the worker followed by an acquire by the manager, while a departure is modeled as a release by the manager followed by an acquire by each of the workers. Therefore, there is a synchronization transfer from each arriving worker process to the barrier manager, and then from the manager to each of the departing workers when they are released.

Like lock releases, arrival messages include consistency information in the form of a `send_set`. Unlike lock releases, arrival messages are not preceded by a request from the manager containing the manager's vector timestamp. Since the worker's knowledge of the manager's vector timestamp may be out of date, the `send_set` sent on the arrival message may be larger than necessary. The manager simply discards any incoming write notices that it has already seen.

Release messages also contain `send_sets`, but these will be no larger than necessary because the `send_sets` on arrival messages implicitly include the workers' vector timestamps.

### Multiple Writers

Figure 2.14 shows an example where two processors, $P_1$ and $P_3$, modify logically distinct pieces of data, $x$, and $y$. Yet because the system tracks accesses at the level of virtual memory pages and these pieces of data are co-located on the same page,

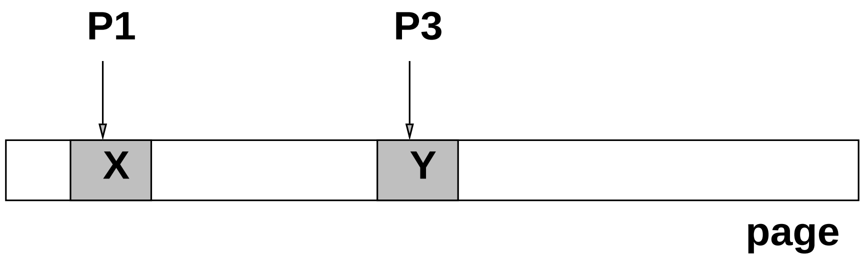


**Figure 2.14** False Sharing

the system must assume that the accesses are competing. In a conventional DSM that uses a single-writer, multiple-reader protocol, the page may "ping-pong" across the network because both processors will simultaneously try to gain sole ownership of the contested page.

False sharing does not cause ping-ponging under any of the protocols discussed in this chapter because they all allow multiple concurrent writers. Figure 2.15 shows how this example might be handled under a lazy protocol. Processes $P_1$ and $P_3$ are again modifying logically distinct pieces of data on the same page. Although $P_1$ and $P_3$ never communicate directly, $P_2$ can construct a copy of the page that includes all of the changes by applying the diffs that summarize the two modifications. There is no communication between the two, and yet summarizing the modification as diffs allows $P_2$ to update its copy of the page to reflect modifications made in both of the other processes merely by applying the diffs.

We can infer from the lack of synchronization between $P_1$ and $P_3$ that the sharing in this example is false sharing, because otherwise such unsynchronized accesses would constitute a data-race, and all programs that run on our system are required to be data-race-free.

Since all programs are known a priori to be data-race-free, modifications made to a page that is falsely shared are not ordered by $\xrightarrow{\text{hb1}}$. The modifications must

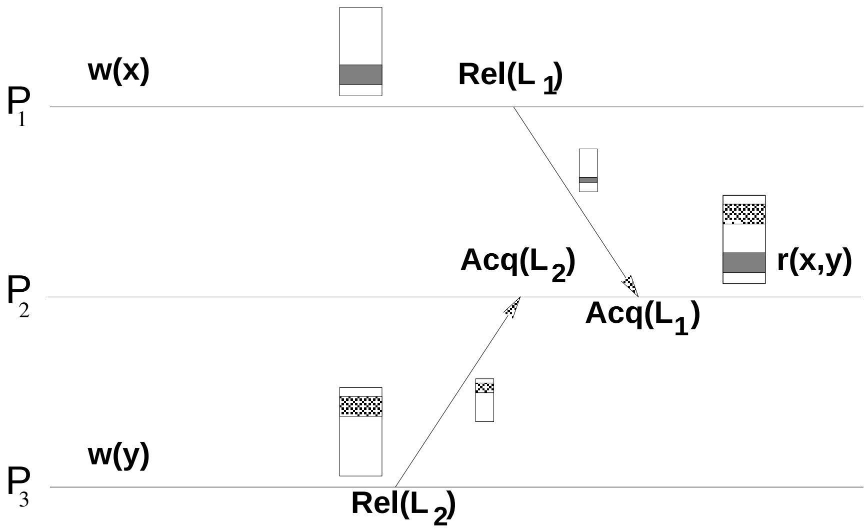


**Figure 2.15** Multiple Writers: X,Y on same page

be non-overlapping because overlapping concurrent modifications would constitute a data race. Since the diffs are known to be non-overlapping, they can be applied to $P_2$'s copy of the page in either order without changing the final result.

### Lazy Diff Creation

We have so far implied that diffs for each dirty page are immediately created upon a lock release or barrier arrival. In fact, diffs can be created far less frequently without violating correctness.

We reduce the number of diffs created through the use of *lazy diffing*. Lazy diffing means that only a write notice is created at the time of the release; diff creation is deferred until a subsequent request for the diff or until a write notice for the same page is received from another process. Until a diff is actually created, additional modifications to the page continue to be accumulated, and the eventual diff includes all of the modifications, even those that occurred after the first release operation. The gain in performance can be considerable, because the eventual diff may include modifications that would have been split over several separate diffs in a system that did not support lazy diffing.

Reducing the number of diffs created saves considerable overhead, as diff creation in our initial environment averages about 800 $\mu$secs a piece. A secondary benefit is that lock acquisitions are faster because grant messages are not delayed until diffs are created. Finally, reducing the number of diffs can reduce the overall amount of data sent over the network, because programs with significant temporal locality often overwrite the same locations many times, and a lazy diff will only include the last values written to each location.

An obvious area for improvement would seem to be in combining multiple diffs of a single page into a single diff over and above the lazy diff mechanism. This would save bookkeeping, diff storage overhead, and possibly network communication. Unfortunately, diff combining is possible only in very specific circumstances.

A simple example of the problem with diff combining is illustrated by Figure 2.16. $P_1$ combines diffs describing the results of operations $w(x)1$, $w(y)1$, and $w(x)2$ into a single diff. In order to satisfy its read of $x$, $P_2$ must apply the combined diff. The combined diff, however, not only updates $P_2$'s view of $x$, but it overwrites $y$ as well. Therefore, $P_2$'s subsequent read of $y$ will return 1, instead of the correct value of 2.

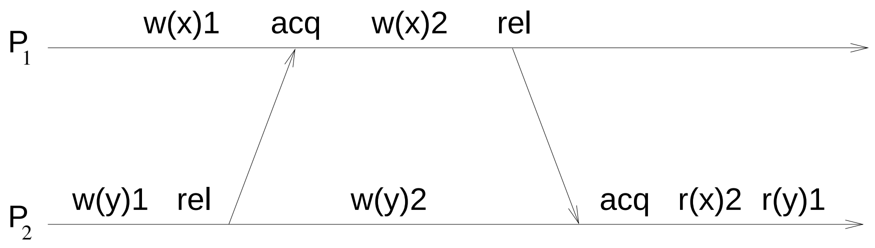


**Figure 2.16** Bad diff combine: $x, y$ same page

In general, diffs can only be combined in the absence of learning of new, possibly interleaving, diffs to the same page by other processors. One possible mechanism would be to allow a process, $P_1$, to combine a run of diffs $\mathit{diff}^{i}_{1,x}$ through $\mathit{diff}^{j}_{1,x}$, where $\mathit{diff}^{i}_{1,x}$ is the $i$th diff of page $x$ created by processor $P_1$, only if there is no $\mathit{diff}^{k}_{2,x}$ where $\mathit{diff}^{k}_{2,x} \xrightarrow{\text{hb1}} \mathit{diff}^{j}_{2,x}$ and not $\mathit{diff}^{k}_{2,x} \xrightarrow{\text{hb1}} \mathit{diff}^{i}_{2,x}$. Such a mechanism can be implemented by retaining the twin even after a diff has been created, and using the twin to create subsequent diffs *until that processor receives another write notice for the same page from another processor*. Drawbacks of this technique include the overhead of retaining twins, and the increasing size of the combined diffs, and slightly complicated bookkeeping in the routines that manage diffs.

A second possibility is to discard diffs if they are completely over-written by later diffs to the same page.

Our implementation currently uses neither of the above optimizations because our results show that in our environment, per-message overhead is high enough that any possible savings from diff combining are unlikely to greatly affect overall performance.

### Garbage Collection

In an eager system, garbage collection is unnecessary because diffs are immediately flushed to every other copy in the system, and therefore no longer needed. In a lazy system, diffs need to be retained until it is clear that the diffs will no longer be requested, i.e. the diffs have already been sent to every processor, or all copies of the page have the diff applied.

In TreadMarks, any process may request a garbage collection from the manager, which initiates the algorithm at the next global barrier, piggybacking a `repo_start` message on the barrier releases. At receipt of a `repo_start` message, worker processes validate every page that is not in an `UNMAPPED` state. Once all pages are either `READ_ONLY` or `UNMAPPED`, a `repo_complete` message is sent back to the manager. After `repo_complete` messages have been collected from all workers, the manager sends out `repo_release` messages and the system continues on as before. Section 3.3.2 shows that even this brute force approach costs little in terms of processor cycles.

### 2.3.2 Lazy Hybrid

The aim of the lazy hybrid protocol (LH) is to reduce the number of access misses by speculatively moving data before it is requested, rather than only in response to access misses. The central intuition is that processes synchronize in order to share data, and the flow of data is likely to mirror the flow of synchronization.

The LH differs from LI in two respects: diffs may be speculatively appended to lock grant messages, and diffs are flushed prior to barrier arrivals. This section describes only those aspects of LH that differ from LI.

#### Locks

The diffs appended to grant messages are chosen by a heuristic that uses an *approximate copyset* to track access to shared pages by other processes. Copysets are initialized to the page's manager, and other processors are added when either (i) another processor requests either a diff of the page or the page itself from the local processor, or (ii) a write-notice describing a modification to that page by the remote processor is seen by the local processor. Processors are never removed from copysets.

The assumption behind the heuristic is that programs usually have significant temporal locality, and therefore any page accessed by a process in the past is likely to be accessed in the future. The heuristic therefore selects diffs of pages that the copyset indicates have been accessed by the lock destination in the past. The search for diffs is limited to diffs corresponding to write notices in the `send_set`. Any diffs that do not fit into the lock grant message are sent in additional unacknowledged messages. The additional messages can be unacknowledged because they do not contain consistency information, and therefore do not violate correctness if lost.

#### Barriers

Prior to barrier arrivals, processes under the LH protocol *flush* likely diffs to all other processes in the system. Figure 2.17 shows pseudo-code for the flush procedure.

The flush procedure creates a `send_set` analogous to the lock `send_set` for each of the other processes in the system. For each of the processes, the flush sends diffs corresponding to all write notices in the `send_set` that were created by the flushing process. The flush operation may take multiple messages, and the messages are not acknowledged because their loss affects only performance, not correctness.

**Lazy Diffing**

The hybrid protocol also uses the lazy diffing mechanism, but to less benefit. Because of lazy diffing, diffs describing recent modifications are unlikely to have been created yet. In order to have the hybrid make a significant impact on performance, LH overrides the lazy diffing mechanism and creates any diffs selected by the heuristics. This can result in many more diffs being created under LH than LI, as well as larger lock acquisition latencies.

### 2.3.3 Eager Invalidate

We base our eager RC algorithms on Munin's multiple-writer protocol [CBZ91]. RC requires all prior ordinary accesses to be globally performed before a subsequent release is performed at any other process. The EI protocol calls a *flush* operation prior to any release. The `flush` operation sends invalidates to all other processes in the system that cache the affected pages. Since the local process may not have complete information on page replication, multiple rounds may be necessary to ensure that the

**foreach** $p \in$ (system processes)
 $\mathit{diffs} \leftarrow \emptyset$
 **foreach** $wn \in$ ( `send_set` to $p$)
  **if** (creator($wn$) = ME) **then**
   $\mathit{diffs} \leftarrow \mathit{diffs} + \mathrm{diff}(wn)$
  **end**
 **end**
 Send diffs unreliably to $p$
**end**

**Figure 2.17** Hybrid Barrier Flush

invalidates are performed on all copies of the page. Once the invalidate messages have been received and acknowledged, the release operation is allowed to proceed.

#### Locks

Under EI, synchronization messages carry no consistency information, and there is no eager equivalent of the `send_set`. Since the consistency action takes place at the release during the flush operation, an acquire consists solely of locating the process that executed the last prior release and transferring the synchronization variable. Lock ownership is determined as under the lazy protocols, taking either two or three messages to return the lock grant.

#### Barriers

Processes arriving at a barrier call the flush operation prior to sending arrival messages to the manager. Barrier messages therefore contain no consistency information.

#### Access Misses

The EI protocol handles page ownership analogously to lock ownership. Each page has a statically assigned manager, but ownership of the page dynamically shifts according to access patterns. Page ownership differs from lock ownership in that the owner of a page does not have an exclusive copy of the page. Instead, the "owner" of a page is defined as the most recent process to request a copy from the manager.

Figure 2.18 shows a state transition diagram for shared pages. Pages are in one of three states: `READ_ONLY`, `READ_WRITE` or `INVALID`. Initially, the page is `READ_ONLY` at the manager and `INVALID` everywhere else. The owner is initially set to the manager. Processes taking an access miss request a copy of the page from the page's manager. If the manager's copy is not `INVALID`, the copy is returned directly to the missing process. Otherwise, the manager forwards the request to the current owner, which responds with a copy of the page. In either case, the process that incurred the access miss is designated the new owner, and the state of the new owner's copy is set to either `READ_ONLY` or `READ_WRITE`, depending on the type of the access.

When a write is attempted to a `READ_ONLY` page, a copy (twin) of the page is created and the page's state is set to `READ_WRITE`. A flush operation discards all twins, sets the twinned pages' state back to `READ_WRITE`, and sends invalidation messages to all other processes that have copies of the affected pages. If an invalidate for

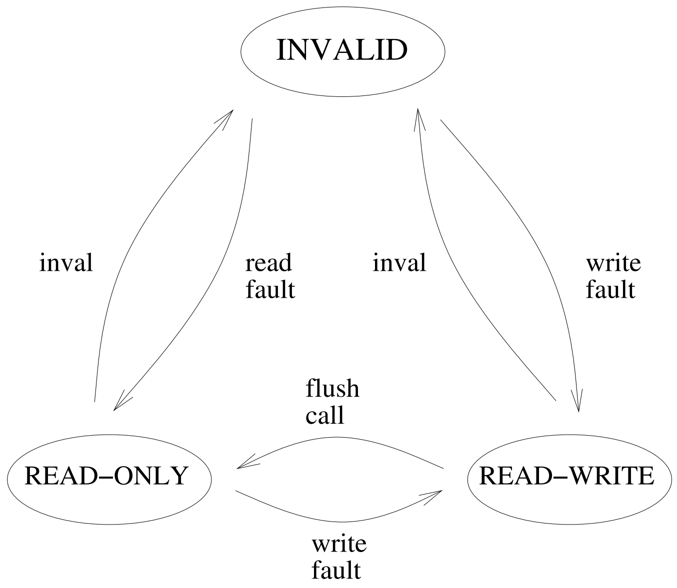


**Figure 2.18** RC Page State Transitions

a `READ_WRITE` page is received, a diff of the page is created and returned to the invalidator on the invalidate reply, and the local copy of the page becomes `INVALID`.

### The `flush` Operation

The `flush` operation ensures that any prior local modifications are performed globally. The modifications are performed by sending invalidations to all other processes that cache the affected pages. Since local information about page caching may be out of date, each process appends their copyset for the invalidated pages to their acknowledgment. The returned information is used to direct additional rounds, if necessary.

If an invalidate is received for a page that is locally in state `READ_ONLY`, the page state is set to `INVALID` and read permission is removed. If the page was in state `READ_WRITE`, a diff is created to describe local modifications, and then the page state and permissions are changed as for `READ_ONLY`. The diff is returned on the acknowledgment message. If there is not sufficient space in the acknowledgment message to copy the diff, an indication that the diff is available is returned instead. The invalidating process then explicitly requests the diff(s) with another pair of messages.

The protocol becomes greatly complicated when multiple processors attempt to concurrently invalidate the same pages. One of these processes may be unaware of the other until after `flush` completes, and either or both of the processes may receive diffs from other processes. In this case, the protocol resorts to flushing all diffs to all processes in order to insure that at least one process retains a valid copy of the page, and that diffs are applied to every copy that remains. We observed this global system update occuring in only one of our programs.

Strictly speaking, this implementation *performs* ordinary accesses earlier than necessary, thereby reducing the opportunity for overlap of communication and computation. Definition 2.2 only requires ordinary accesses to be performed when a subsequent release operation is performed. By definition, however, a release is not performed until the same lock is acquired by another process. Therefore, the `flush` operation is not required to complete until the lock is requested by another process. However, allowing the flush to operate concurrently with the computation would either increase lock acquisition latency when `flush` requires multiple rounds, or incur additional signal handling overhead. In either case, the protocol is greatly complicated.

## 2.4 Correctness

This section presents a proof that LI guarantees sequentially consistent executions for all data-race-free programs, where a sequentially consistent execution is an execution, or a specific run, of a program that could have been produced by an SC system. The proof for LH is identical.

This proof is based on a result by Adve [Adv93] that any system meeting a specific set of sufficient conditions guarantees to produce only sequentially consistency executions. The rest of this section re-caps some necessary definitions and then shows by inspection that LI meets the required sufficient conditions, and therefore guarantees SC executions.

DRF1 distinguishes synchronization and non-synchronization operations without imposing any restrictions on *how* they are distinguished. Synchronization operations are either *releases* or *acquires* of synchronization variables. Everything else is a data operation. Intuitively, a program is correct if *enough* operations are distinguished as releases or acquires. In this context, "enough" means that the resulting program has no data races, or is data-race-free (Definition 2.5).

Definition 2.6 and Condition 2.4.1 are from Adve's thesis [Adv93].

**Definition 2.6** A system obeys the *data-race-free-1* memory model if and only if the result of every execution of a data-race-free program on the hardware can be obtained by an execution of the program on sequentially consistent hardware.

Adve showed that any DRF1 system guarantees sequentially consistent executions for all data-race-free programs. Therefore, all that remains is to show that a system using the LI protocol is indeed a DRF1 system.

Condition 2.4.1 specifies sufficient conditions for a system to obey the DRF1 memory model.

**Condition 2.4.1** Hardware obeys the data-race-free-1 memory model if for every execution, $E_{drf}$, of a program, Prog, on the hardware, there is an $\xrightarrow{\text{xo}}$ (and a corresponding $\xrightarrow{\text{hb0}}$*) that satisfy the following conditions:
(i) *Data* - If $X$ and $Y$ are conflicting operations, at least one of $X$ or $Y$ is a data operation, and $X \xrightarrow{\text{hb1}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all $i$.
(ii) *Synchronization* - If $X$ and $Y$ are conflicting synchronization operations, and $X \xrightarrow{\text{hb0}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all $i$.
(iii) *Control* - If Prog is data-race-free, then there exists a sequentially consistent execution, $E_{sc}$, with a well-formed $\xrightarrow{\text{xo}}$ and a corresponding $\xrightarrow{\text{hb0}}$ such that (i) an operation is an $E_{drf}$ iff it is in $E_{sc}$, (ii) for two conflicting operations $X$ and $Y$, such that at least one of them is a data operation, if $X \xrightarrow{\text{hb1}} Y$ in $E_{sc}$, then $X \xrightarrow{\text{hb1}} Y$ in $E_{drf}$, and (iii) for two conflicting synchronization operations $X$ and $Y$, if $X \xrightarrow{\text{hb0}} Y$ in $E_{sc}$, then $X \xrightarrow{\text{hb0}} Y$ in $E_{drf}$.

The notation $X(i)$ refers to the $i^{th}$ sub-operation of $X$. A sub-operation is the performance of the operation at a specific remote site. The notation $\xrightarrow{\text{xo}}$ refers to execution order.

The "data requirement" specifies that conflicting updates are done atomically with respect to one another. This requirement is fulfilled by LI because overlapping

*The ordering $\xrightarrow{\text{hb0}}$ differs from $\xrightarrow{\text{hb1}}$ in that $\xrightarrow{\text{hb0}}$ is specific to a single execution, while $\xrightarrow{\text{hb1}}$ applies to all executions of a given program. To avoid confusion, $\xrightarrow{\text{hb1}}$ is used in the rest of the text. See [Adv93] for the original definitions.

modifications are totally ordered by $\xrightarrow{\text{hb1}}$, and diffs are always requested and applied in an order consistent with $\xrightarrow{\text{hb1}}$.

The "synchronization requirement" specifies that synchronization operations are sequentially consistent with one another. This requirement is met by observing that all LI synchronization operations are SC with respect to one another, because each operation completes atomically before a synchronization variable can be accessed by another process†.

The "control requirement" exists to handle cases where reads determine if an operation will be executed, or which address an operation will access. The control requirement can only be violated without violating data or synchronization requirements if the underlying system reorders operations. Our proposed implementations of LI do not reorder code.

## 2.5 Summary

Table 2.1 summarizes the message counts for lock transfers, barrier arrivals, and access misses for each of the three protocols.

Lock acquires take either two or three messages for all three protocols.

A lock release is an entirely local operation for the lazy protocols, but entails flushing updates for the eager protocols. As the updates must be flushed to all other processes that cache the modified pages, messages may have to be exchanged with all other processes.

The EI protocol takes at most three messages to satisfy an access miss: the page request may need to be forwarded (once) from the owner to a process with a current copy, and then one message is needed to return the page to the missing process. Under a lazy protocol, however, multiple diffs may need to be retrieved as well. In Table 2.1, the *concurrent last modifiers* for a page are the processes that created modifications that do not causally (*via happened-before-1*) precede any other known modifications to that page. In other words, they are the last processes to have modified the page. In the absence of false sharing, this set includes only a single process. This set potentially includes every other process in the system in the presence of false sharing.

†A barrier can be viewed as a series of $n-1$ release-acquire transactions between system processes and the manager process, followed by $n-1$ release acquire transactions from the manager to the other processes. The ordering of the barrier sub-operations is the order in which barrier arrival messages are received and the order in which barrier release messages are seen.

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

$m$ = # concurrent last modifiers for the missing page
$c$ = # other cachers of the page
$n$ = # processes in system
$p$ = # pages in shared space
$u = \sum_{i=1}^{n}$(# other procs caching pages modified by $i$)

**Table 2.1** Shared Memory Operation Message Costs

Finally, the total base cost of barrier operations for all the protocols is $2(n-1)$ messages, which represents $n-1$ arrival messages followed by $n-1$ barrier release messages. Additionally, if pages are concurrently modified by multiple processes, the LH protocol requires updates to be flushed prior to arriving at the barrier. However, the diffs are not sent reliably, and so the number in the table does not reflect acknowledgments.

EI's invalidations can also increase lock acquisition latency because releases can not be performed until invalidations have been sent and acknowledged. Neither of the lazy protocols require consistency information to be sent to other processes, but 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.

Since the lazy protocols usually exchange data in the form of diffs, the total amount of data exchanged is usually less than for EI. A potential advantage of EI is that it creates diffs only in uncommon situations. However, as Chapter 3 will show, the cost of the diffing mechanism is usually minor compared to communication costs.

None of the protocols is overly complex to implement. The lazy consistency management routines are completely implemented in 1200 lines of commented code. Support for the hybrid protocol requires an additional 288 lines, primarily for the barrier flush routines. The eager protocol's invalidate and flush routines are implemented in 800 lines of code.

Table 2.2 summarizes anticipated tradeoffs among LI, LH, and EI.

| Prot. | Lock Latency | Remote Access Faults | Msgs | Data | Diffs | Prot. Complexity | Prot. Cost |
|---|---|---|---|---|---|---|---|
| EI | Low | High | High | High | Low | Low | Low |
| LI | Low | Medium | Medium | Low | Medium | Medium | Low |
| LH | Medium | Low | Low | Medium | High | Medium | Low |

**Table 2.2** Protocol Tradeoffs

Overall, LI and LH tend to require less communication than EI, especially for programs that use locks. Their primary advantage is that communication is limited to the two synchronizing processes during lock transfers. A release in an eager system may require invalidations to be sent to processes otherwise uninvolved in the synchronization. In combination with false sharing, these invalidations can then cause additional access faults.

LH should have slightly higher lock acquisition latency due to the overhead of creating and sending diffs with lock grants. The extra diffs should eliminate some access misses, and hence reduce the overall number of messages.

# Chapter 3

# Performance

This chapter presents an evaluation of the two lazy release consistent protocols: lazy invalidate (LI) and lazy hybrid (LH). The protocols were implemented in the TreadMarks distributed shared memory system. We used eight programs in the evaluation: n-body simulation (Barnes), FFT, Integer Sort (IS), Linkage Analysis (ILINK), Mixed-Integer Programming (MIP), Successive Over-Relaxation (SOR), the Traveling Salesman Program (TSP), and a molecular dynamics simulation (Water).

Our evaluation of the LRC protocols has three components. The first is a comparison of the performance of the lazy protocols with the performance of an eager invalidate (EI) protocol. We use EI because studies [DKCZ93, KCZ92] have shown that EI consistently outperforms eager update protocols, and previous studies [CBZ91, Car93] showed that eager update protocols consistently outperform conventional DSM protocols. All of the protocols were implemented in the same system, running on the same hardware, and incur the same costs for communication and virtual memory primitives. The comparison therefore measures the extent to which LRC improves on the state of art. Seven of the eight programs perform better with the lazy protocols. Four of the eight programs, Barnes, IS, MIP, and Water, improve by at least 18%. TSP performs 12% better, SOR performs 3% better, and FFT actually performs 18% worse. This comparison shows the broad range of applications and access patterns for which the lazy protocols are able to improve performance.

We then break the application execution times into four major categories: application, Unix primitives, TreadMarks, and idle time. The central goal is to measure the additional cost of executing the slightly more complicated lazy protocol code, and to determine whether this additional cost is significant. We found that no application spent more than 4.3% of its time executing protocol code, and the average over all applications and all of the protocols was only 1.2%[‡]. By contrast, the applications spent more than 10% of their time executing Unix primitives, and on the order of

[‡]Split into lazy and eager overheads!

20% of their time was idle time, which is usually dominated by the latency of remote requests. This latter category is therefore tightly tied to the cost of the underlying communication primitives because the primitives usually take much longer than actually serving a request. Our conclusion, for this environment, is that the cost of executing protocol code is negligible compared to the cost of network communication. Moreover, it is the *software overhead* incurred by the operating system in sending and receiving messages that dominates communication cost. For our current system, "wire time" is an insignificant contributor to overall performance. Therefore, the impact the protocols have on the amount of required communication is far more important than the direct protocol execution cost.

Finally, we used a profiling tool, `vt` [BL92], to rewrite our executables to add instrumentation code that estimates execution time in terms of processor cycles. With the instrumented code and a small library, we were able to use TreadMarks as a parallel simulator to explore the changing relationships between the consistency protocols as we varied network and operating system parameters. This simulation is more accurate than our previous studies because it executes the actual protocol code. We found that the performance gap between the lazy and eager protocols is highest in systems that have large communication costs, either because of low bandwidth or expensive operating system primitives. The lazy protocols outperformed the eager down to software overheads ten to twenty times less than our current system.

## 3.1 Experimental Environment

### 3.1.1 Hardware Platform

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, although the effective bandwidth is approximately 30 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. The machines are also connected by a 10-Mbps Ethernet. Unless otherwise noted, all performance numbers describe 8-processor executions on the ATM LAN using the low-level adaptation layer protocol AAL3/4.

### 3.1.2 Basic Operation Costs

Table 3.1 shows the cost of sending various sized messages over the AAL3/4 sockets. Receipt times are comparable. The minimum roundtrip time is 450 $\mu$seconds. Sending each of the two messages takes 80 $\mu$seconds, receiving each of the two messages takes a further 80 $\mu$seconds, and the remaining 130 $\mu$seconds are divided between wire time, interrupt processing and resuming the process that blocked in receive. Using a signal handler to receive the message at both ends increases the roundtrip time to 620 $\mu$seconds.

The minimum time to acquire a non-local free lock is 827 $\mu$seconds if the manager was the last process to hold the lock, and 1149 $\mu$seconds otherwise. In neither case does the reply message contain any write notices (or diffs). Lock acquisition cost increases in proportion to the number of write notices that must be included in the reply message. The minimum time to perform an 8 process barrier is 2186 $\mu$seconds. A non-local page fault, to obtain a 4096 byte page from another process, takes 1956 $\mu$seconds.

## 3.2 Applications

We used eight applications in this study: `Barnes`, `FFT`, `ILINK`, `IS`, `MIP`, `SOR`, `TSP`, and `Water`. The applications vary widely in their complexity. For example, `SOR` and `TSP` are relatively simple, while `ILINK` and `MIP` each consist of more than ten thousand lines of code. The programs come from several different sources. `Barnes` and `Water` come from the Stanford Parallel Applications for Shared Memory (SPLASH) benchmark suite [SWG91]. `FFT` and `IS` come from the NAS benchmark suite [BBLS91]. `SOR` and `TSP` were developed at Rice University. `ILINK` was derived from a sequential program used by geneticists worldwide [LLJO84, CIS93, DSC$^+$94]. `MIP` was developed by researchers from the Department of Computational and Applied Mathematics at Rice University. Both `ILINK` and `MIP` are production codes used to solve *real* scientific and commercial problems that may require days or weeks of computation. The next several sections describe these programs in detail.

| Message Size (bytes) | 8 | 128 | 1024 | 4096 | 9188 |
|---|---|---|---|---|---|
| Time ($\mu$secs) | 80 | 127 | 156 | 441 | 1367 |

**Table 3.1** Cost of sending messages

### Barnes-Hut (Barnes)

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 each body is modeled as a point mass and exerts forces on all other bodies in the system. The complexity is $O(n^2)$ in the number of bodies if all pairwise forces are calculated directly. Barnes-Hut uses a hierarchical tree-based method that reduces the complexity to $O(n \log n)$. The numbers presented are for a run using 4096 bodies.

### 3-D Fast Fourier Transform Benchmark (FFT)

This benchmark numerically solves a partial differential equation using forward and inverse FFT's. Assuming the input array A is $n_1 \times n_2 \times n_3$ and organized in row-major order, we distribute the array elements along the first dimension of A. For any $i$, all elements of $A[i,*,*]$ are contained within a single process. 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. Each process computes its portion of the array without any communication. Processes require data from other processes only when they are ready to work on the $n_1$-point vectors in the first dimension. This means that only one transpose is needed for each iteration of the 3-D FFT.

With N processes, every transpose requires each process to send 1/N of its data to every other process and to receive 1/N of its data from each of the other processes. Since the array is often several megabytes or larger, the time spent on the transpose can be the limiting factor on overall performance. We ran the tests with array dimensions of 64x64x32.

### ILINK

ILINK [DSC+94] is a parallel version of FASTLINK 1.0 [CIS93], which is an improved version of LINKAGE [LLJO84]. Genetic linkage analysis is a statistical technique that uses family pedigree information to map human genes and locate disease genes in the human genome. The fundamental goal in linkage analysis is to compute the recombination probability, which is the probability that a recombination occurs between two genes.

ILINK searches for a maximum likelihood estimate of the multi-locus vector of recombination probabilities of several genes. 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 [DSC+94]) 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, so the approach in Figure 3.1 was used instead. This version uses a local copy of the genarray, called `gene`, to temporarily hold updates to the global array. They are eventually merged into the final copy after synchronization.

ILINK's input consists of data on 12 families with autosomal dominant nonsyndromic cleft lip and palate (CLP) [HWC+93].

**Integer Sort (IS)**

This application 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 program is parallelized by dividing the array among the processes. 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 between processes. Processes synchronize through barriers between rankings, and through locks during rankings.

```
For each pedigree
    For each nuclear family
        Split up double loop over possible genotypes for each parent
        For each process
            Do updates to gene for assigned rows
        Synchronize processes to sum updates together into genarray
            using a barrier
```

**Figure 3.1** Parallel Linkage Computation

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.

### MIP

Mixed integer programming (MIP) is a version of linear programming where some or all of the variables are constrained to have integer values, or sometimes just the values 0 and 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 can not currently be solved.

The MIP code uses a *branch-and-cut* approach. The integer problem is first relaxed to a linear programming problem, which will generally lead to a solution in which some of the integer variables take on non-integer values. The next step is to pick one variable 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 solution. 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. In order to expedite this process, the algorithm uses a technique called *plunging*, essentially a depth-first search down the tree to find an integer solution and establish a bound as quickly as possible. The input set used in our runs was `misc05.mps`.

### 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 a function of the values of neighboring elements. In this case, the function is an 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 process. Exchange of data between processes is therefore limited to those pages containing rows on the edge of the chunks.

Barriers are used to synchronize all processes at the end of each half-iteration. Our input set is 2000 x 1000 floating point numbers.

### Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) finds 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 from one city to another has an associated weight. The cost of a tour is the sum of the weights of each leg of the tour.

We use a branch-and-bound algorithm. A global value known as `best_tour` contains the cost of the least expensive complete tour found at any given time. Searches are pruned by abandoning any partial tour whose cost exceeds `best_tour`. Each process repeatedly acquires the queue lock, removes and resolves partial tours until it has a partial tour that is complete enough to solve locally, releases the queue lock, and solves the partial tour locally. The queue is a sorted heap, sorted in inverse order of a lower bound on their total cost. The lower bound is calculated using a fast greedy algorithm. Tours removed from the head of the queue are therefore the most promising queues and therefore the chances that later tours will be pruned before being solved is increased. When a partial tour is within a constant number of cities from being complete, the searching of the tour is completed through local recursion. The shared data structures include the priority queue, which contains pointers to the actual tours, a stack of unused tours that may be re-used, and `best_tour`. Individual locks protect access to the work queue, the unused tour stack, and `best_tour`.

Several factors make this program less than ideally suited for our benchmark. First, access to the priority queue is centralized. Secondly, the use of indirect structures such as the priority queue means that multiple faults will occur in taking tours out of the queue. Finally, read access to `best_tour` is not synchronized, and therefore the program is not *data-race-free* (see Section 2.2.4). The reason for this is that access to `best_tour` would become a serious bottleneck if a lock had to be acquired each time a read occurs. The program works as it is, but RC systems in general will not generate correct results for programs that are not properly labeled (data-race-free). Although processes may see out of date values of `best_tour`, the results are always correct because `best_tour` is monotonically increasing. Out of date values for `best_tour` may result in pruning less work than if `best_tour` were always consistent. Since lazy protocols delay propagating consistency information, processes running on a lazy system may do more work than processes running on eager systems because they receive new `best_tour` values later than their eager counterparts, and hence are not able to prune local tours as quickly. We ran TSP on a nineteen city input set.

### Water

Water is a slightly modified version of the Water program from SPLASH [SWG91]. Our version differs from the standard by condensing multiple lock acquisitions into single acquisitions, thereby greatly reducing synchronization overhead. Only nine lines of code were affected by the change.

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.

The main shared data structure in Water is a large, one-dimensional array of molecules called `VAR`. Equal contiguous chunks of the array are partitioned to each process. Each molecule is represented by a 600-byte record that includes data describing the molecule's displacement, the first six derivatives, and computed forces. Approximately seven molecules fit on each virtual memory page.

Each time-step of Water uses several barriers. However, synchronization behavior is dominated by the phase that calculates inter-molecular forces. Ignoring the cutoff radius for the time being, each process needs to compute $\frac{N^2}{2p}$ interactions, where $N$ is the number of molecules and $p$ is the number of processes. Each interaction includes reading the particle positions of the interacting molecules and updating force equations for each molecule.

In the original Water program, a lock has to be acquired for each participating molecule in an interaction.

We simulated 343 molecules for 5 steps.

**Application Diversity**

Our applications differ greatly in the type and frequency of synchronization, the degree of sharing, the degree to which the data domain of a particular process changes over the length of a program execution, and the granularity of shared accesses.

Table 3.2 summarizes the applications and their input sets. `Syncs_per_second` is the total synchronization rate for an eight processor run under LI.

SOR is a simple, barrier-based, numeric application that distributes data evenly among the processes, and accesses the data in predictable patterns. Another barrier application, ILINK, synchronizes infrequently but has a complex and dynamic sharing pattern. Barnes and FFT are also barrier based programs, but Barnes' major data structures are trees, and FFT periodically undergoes a phase change in which processes exchange data and begin to work on entirely different sections of the main array.

IS and Water use both locks and barriers. However, Water sends messages at a rate more than four times IS's rate (See Table 3.3) because it shares data more finely and synchronizes more often.

Finally, TSP and MIP both use branch and bound algorithms and synchronize exclusively through locks. However, MIP synchronizes more than 30 times as frequently as TSP.

In total, the programs of our application suite represent a wide variety of algorithms, parallel programming styles, and data access granularities. The diversity of

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

**Table 3.2** Application Suite

our application suite ensures that the results of this study are representative of a large class of programs, rather than being specific to a single type.

## 3.3 Comparative Evaluation

This section evaluates the performance of LI, LH, and EI. We trace differences in performance to application and protocol characteristics that make specific protocols more or less suited to particular types of applications.

### 3.3.1 Results

Figures 3.7 to 3.9 present speedups for the applications for all three protocols using up to 8 processors. Table 3.3 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_Faults` is the number of access faults per second that required remote com-

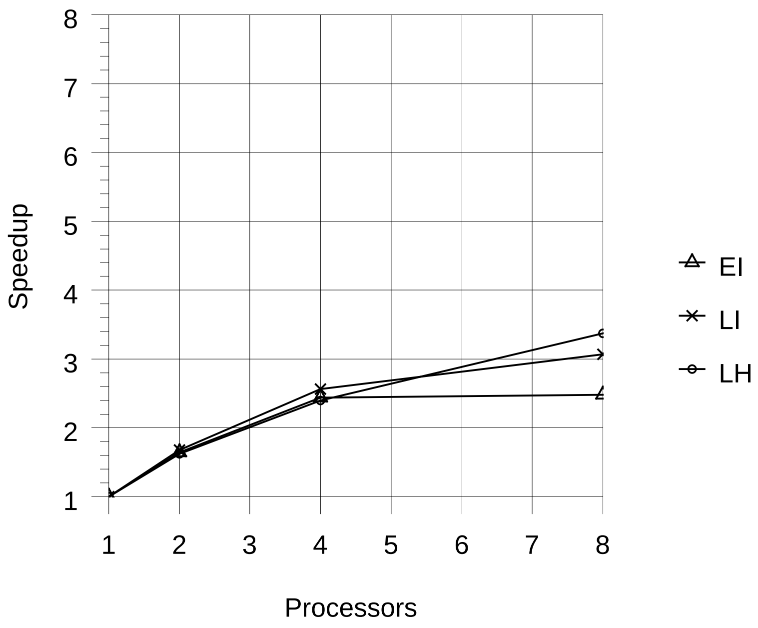


**Figure 3.2** Speedups for Barnes Hut

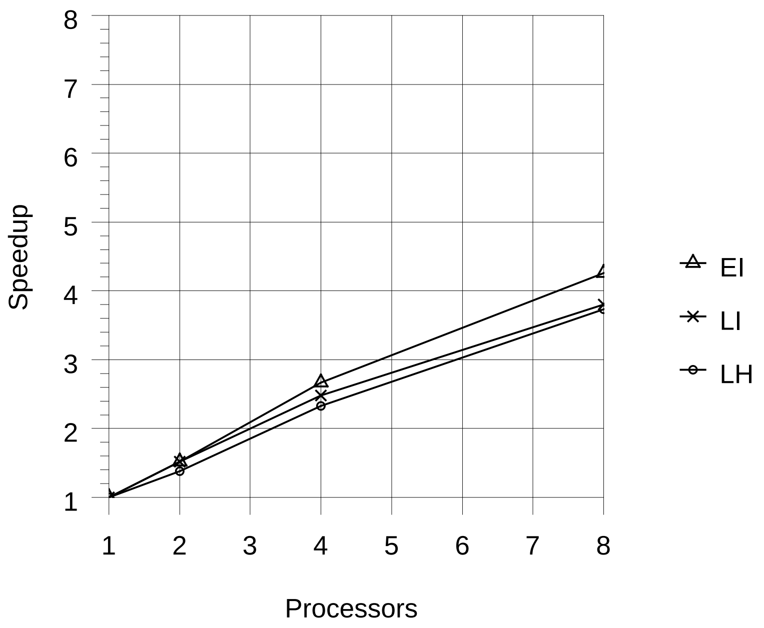


**Figure 3.3** Speedups for FFT

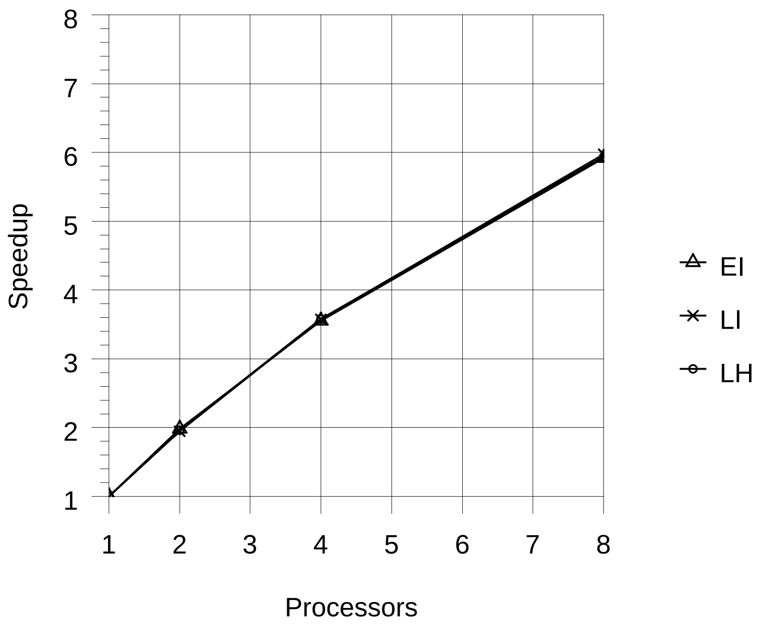


**Figure 3.4** Speedup for ILINK

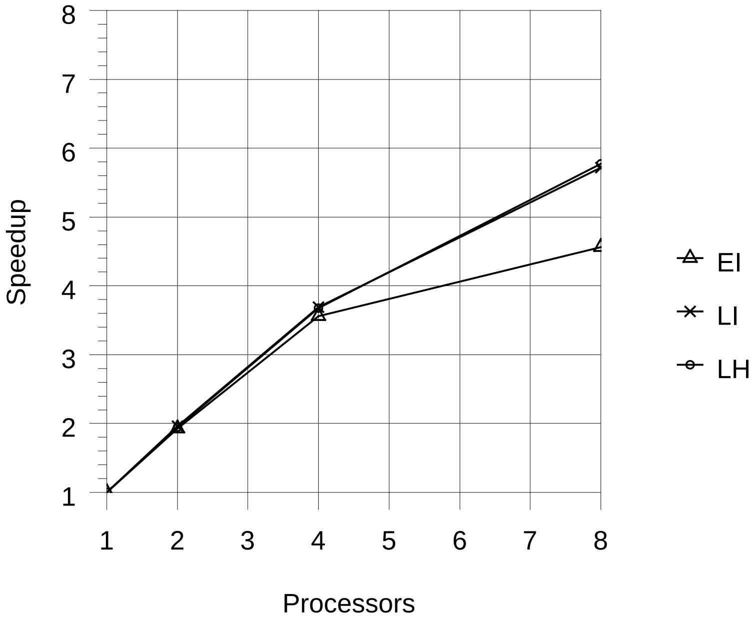


**Figure 3.5** Speedups for IS

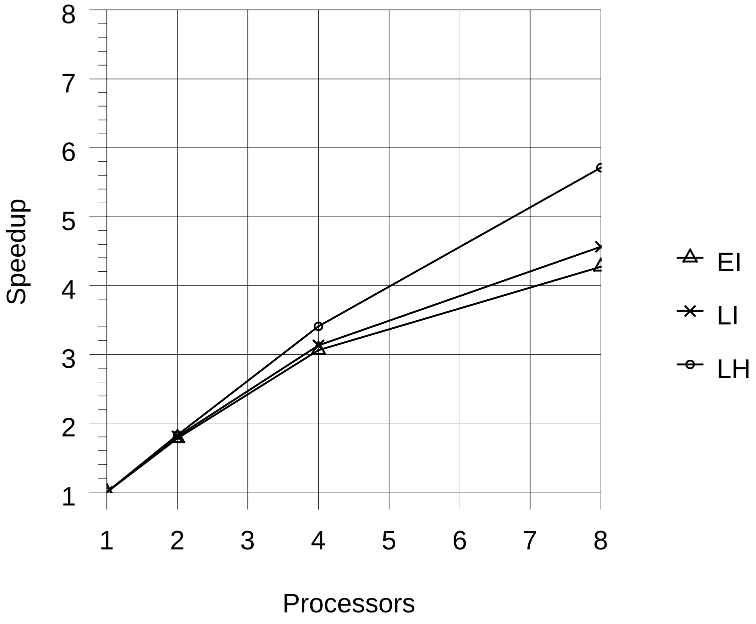


**Figure 3.6** Speedups for MIP

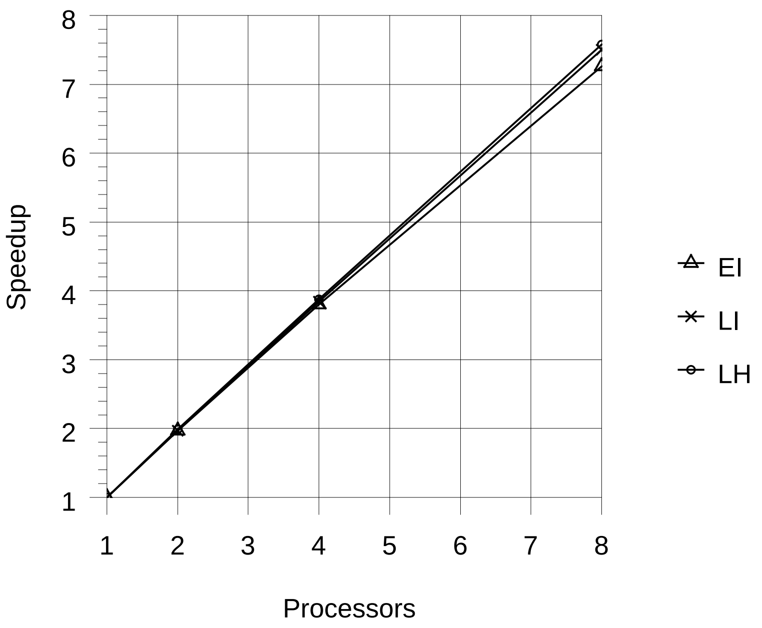


**Figure 3.7** Speedups for SOR

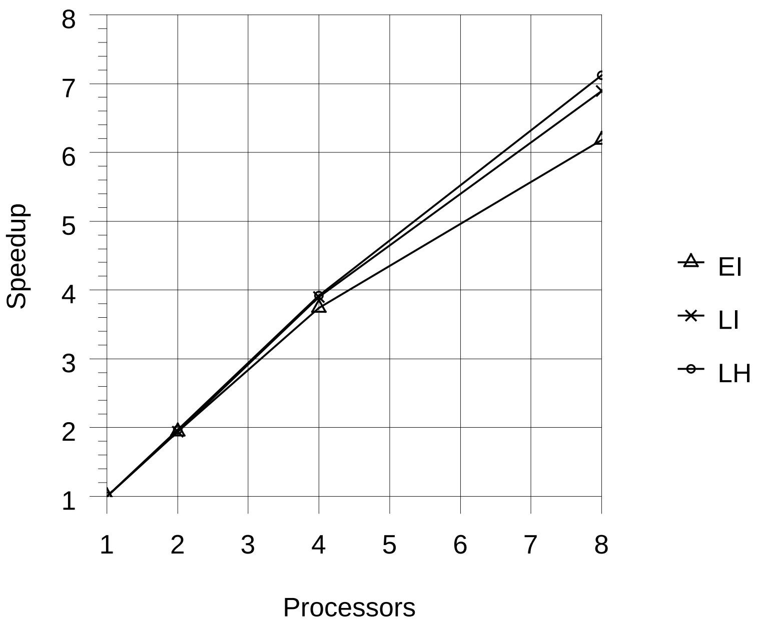


**Figure 3.8** Speedups for TSP

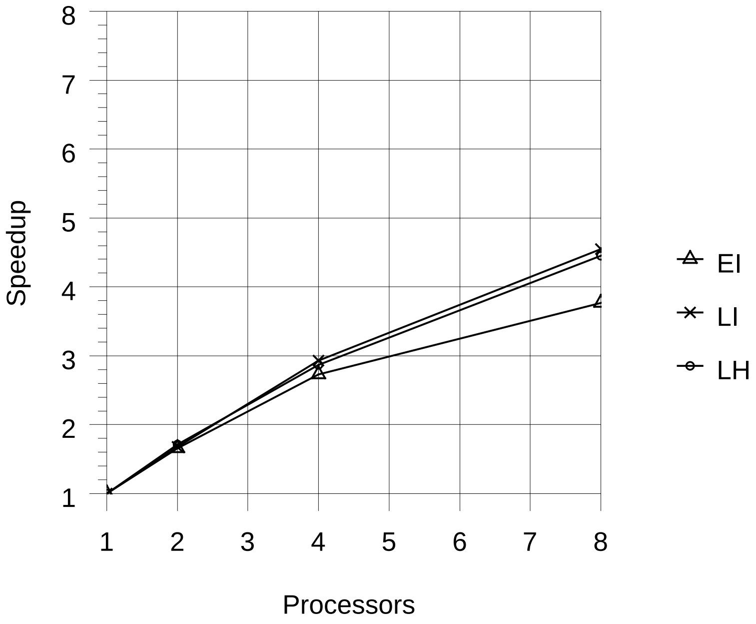


**Figure 3.9** Speedups for Water

munication. Finally, `Diffs_Created` is the rate at which diffs were created in the system.

### Barnes

The speedup for Barnes is 2.5 for EI, 3.0 for LI, and 3.4 for LH. The poor performance is largely due to 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.

LH is able to reduce the overall number of diffs requested by more than half. However, Table 3.3 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

| Program | Prot | Run Time (secs) | Total Msgs (per sec) | Data (Kbytes per sec) | Access Faults (per sec) | Diffs Created (per sec) |
|---|---|---|---|---|---|---|
| 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 |
| 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 |
| 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 |
| 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 |
| 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 |

**Table 3.3** Lazy and Eager Rate Statistics

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 minimal.

EI suffers almost 50% more access faults than LI, and under EI, each of these faults requires an entire page to be retrieved across the network.

### FFT

The speedup for FFT is 3.7 for LH, 3.8 for LI, and 4.2 for EI. 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.

A more serious problem is that in some cases several of these full-page diffs are applied consecutively to the same page. This occurs because data in FFT is *migratory*. During each iteration, a complete transpose is done on the FFT data, and processes are assigned new portions of the array to compute. Before accessing a newly assigned portion of the array after a transpose, processes must first apply diffs describing all previous modifications to that portion. If "ownership" of page $p$ has cycled through three different processes prior to $p$ being assigned to process $P_4$, $P_4$ must first apply diffs describing the modifications to $p$ performed by $P_1$, $P_2$, and $P_3$, even if each diff completely overwrites the previous diffs.

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.

Table 3.3 show that LH sends more messages per second than LI. The extra LH messages are barrier flush messages, most of which are useless because the transposes make past behavior a poor predictor of future accesses.

### ILINK

The speedup for ILINK for all three protocols is 5.9. Overall, ILINK achieves less than linear speedup because several sections of code are executed sequentially rather than in parallel, and the algorithm has an inherent load balancing problem [DSC+94]. It is not possible without significant computation and communication to predict in advance whether the set of iterations distributed to the processes will result in each process having an equal amount of work. Consequently, speedups are somewhat lower than one would expect based on the communication and synchronization rates. The load balancing problem is made worse by the cost of the DSM's network communication.

ILINK is another barrier-only program, so LI has few opportunities to improve performance. LH reduces the miss rate and overall communication rate substantially, but again is unable to affect overall performance.

Table 3.3 shows that EI sends more than four times as much data as LI. The extra data is the result of sending entire pages rather than the diffs used for the lazy protocols. Performance remains virtually identical across all of the protocols because there is very little communication even under EI.

### IS

The speedup for IS is 4.6 for EI and 5.7 for the lazy protocols. 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. These reads often cause access misses 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 access misses 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 3.3 shows that LH sends significantly more messages than LI. The extra messages are barrier flushes that, like in FFT, are often useless. The messages may be useless because many of the diffs communicated by the barrier flushes have already been received via lock grant messages.

### MIP

The speedup for MIP is 4.3 for EI, 4.6 for LI, and 5.3 for LH. MIP is a work-queue based program implemented using locks. Hence, the lazy protocols are able to significantly reduce the number of messages and the amount of data communicated. MIP has a large amount of false sharing, and the individual modifications are much less than a page in size. Since EI responds to access misses by sending entire pages, it sends more than ten times as much data as the lazy protocols.

LH performs slightly better than LI because it eliminates more than 40% of the access misses without sending much more data.

### SOR

The speedup for SOR is 7.3 for EI and LI, and 7.5 for LH. SOR's computation to communication ratio is an order of magnitude larger than that of Water. Since most of the communication is one-to-one between neighbors exchanging boundary row elements, the ATM network is able to accomplish much of the communication in parallel. Parallel communication, in combination with a large computation grain size results in near-linear speedup.

The steady state data movement in SOR consists of neighbors exchanging pages containing boundary rows. Assume $p_i$ computes rows $a-k$ to $a$, and $p_j$ computes $a+1$ to $a+k+1$. At the barrier release, $p_i$'s copy of row $a+1$ is invalidated, as is $p_j$'s copy of row $a$. Under LI and EI, then, each will miss on the other's row and require two messages (request, response) to obtain the diff needed to re-validate the page.

The lazy diffing mechanism only creates diffs for approximately three-quarters of the modified pages because of the way data is arranged in memory. Under LH, however, neighboring processes exchange diffs via flushes before arriving at a barrier. The primary benefit from the flushes is that they are unreliable, and so require only a single message. The access misses that occur without the flushes always require at least two messages to handle. However, this gain is partially offset by the cost of creating extra diffs. The extra diffs are created because LH's flushes interfere with the lazy diffing mechanism and do not allow diffs to be combined.

EI has more communication than LI because under EI each process sends invalidate messages to each neighbor upon arriving at a barrier. LI uses fewer messages by passing the invalidations between neighbors via the barrier messages.

### TSP

The speedup for TSP is 6.1 for EI and 7.0 for the lazy protocols. TSP is an application that exclusively uses locks for synchronization. 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 under the lazy protocols 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. Hence, the previous contents are often not needed when a tour record is retrieved from the tour heap. The DSM system, however, reconstructs the last contents of each accessed tour record even though 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 tour lock acquisition latency of over 22 milliseconds (Table 3.3).

EI does a much better job of handling unsynchronized accesses to the `tour_bound` than the lazy protocols, because it invalidates other copies of the page containing the bound, forcing them to retrieve the most recent version. Recall that knowledge of a new bound does not reach a process under the lazy protocol until that process synchronizes, at least indirectly, with the process that created the new bound. However, EI sends more data because misses require entire pages to be retrieved over the network.

TSP performs marginally worse with LH than with LI. Part of the reason is TSP's poor data locality, past behavior is not a good indicator of future access patterns. Nevertheless, LH requires 9% fewer messages than LI for the 19 city problem.

#### Water

The speedup for Water is 3.7 for EI, 4.2 for LH, and 4.6 for LI. Water sends far more messages than the other applications, and almost 70% of these messages for the lazy protocols are lock requests and responses.

The hybrid protocol performs 5% worse than the invalidate protocol because it increases the average lock latency and the total number of diffs created.

The number of diffs goes up because of false sharing between processes accessing adjacent molecules. Almost seven molecules fit on a single page. Lazy diffing often allows LI to delay creating a diff of a page until multiple molecules have been modified, while LH creates diffs more frequently

### 3.3.2 Execution Time Breakdown

We used `qpt` [BL92] to break down the applications' execution times. Figure 3.10 shows the breakdown for each of our applications running on 8 processors, under each of the protocols. "Computation" is the time spent executing application code; "Unix" is the time spent executing Unix system calls and library code (most of this time is 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 requests (diffs, locks) to succeed 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. An obvious conclusion is that *for this environment* the complexity of the protocol

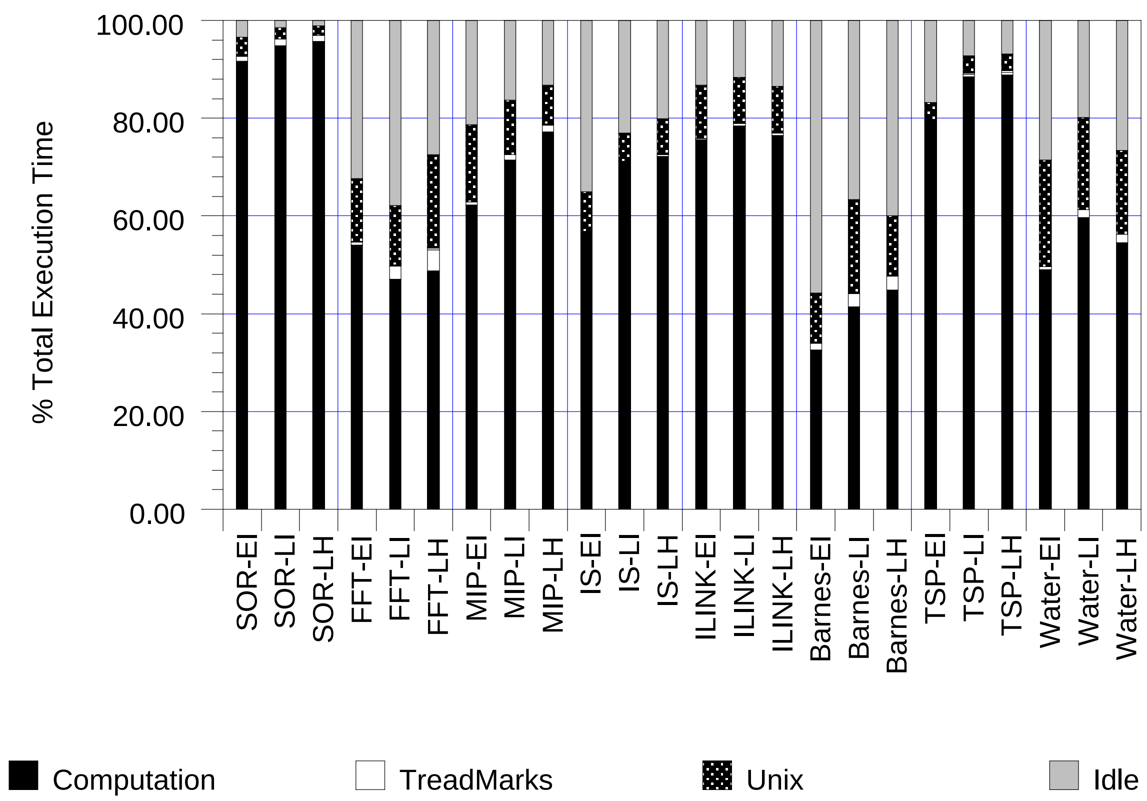


**Figure 3.10** TreadMarks Execution Time Breakdown

matters far less than the number and size of messages required to support the DSM environment.

Figure 3.11 shows our division of Unix overhead into two categories: communication overhead and memory management overhead. 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, EI has the largest Unix communication overhead.

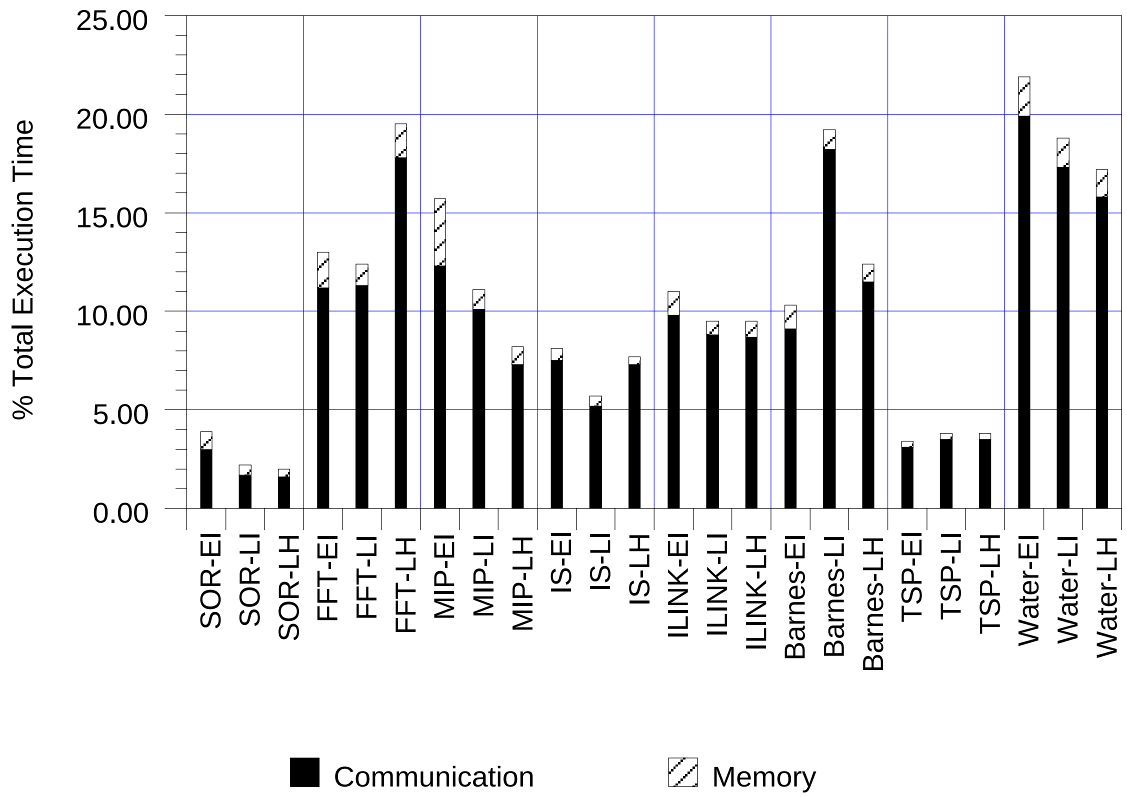


**Figure 3.11** Unix Overhead Breakdown

Figure 3.12 shows a breakdown of TreadMarks overhead into three categories: memory management, consistency, and "other". "Memory Management" is the time spent at the user level detecting and capturing changes to shared 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.

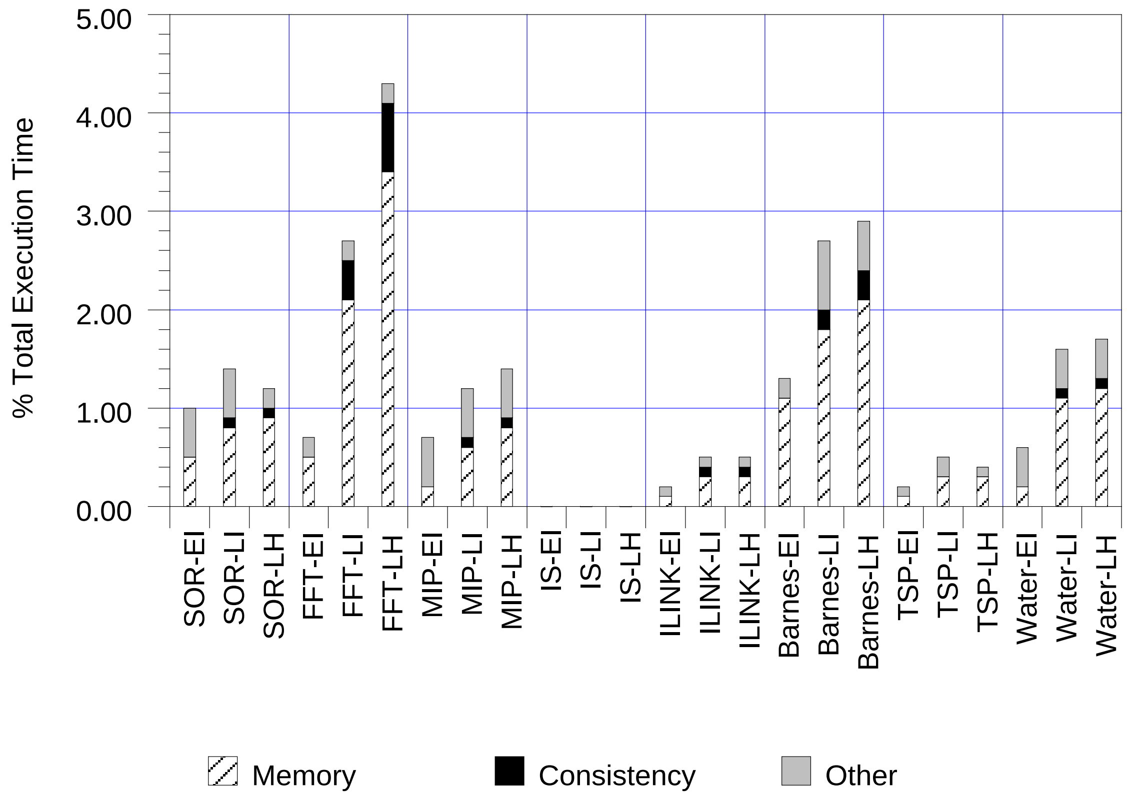


**Figure 3.12** TreadMarks Overhead Breakdown

### 3.3.3 Evaluation of the hybrid heuristic

The effectiveness of the run-time diff selection heuristic is crucial to LH's performance. This section presents a comparison between LH and LP, a hybrid variant that replaces the run-time heuristic with program annotations. The intent of this comparison is to provide a basis for gauging the effectiveness of the heuristic in lock-based programs. We are not concerned in this section about the effectiveness of the heuristic at barrier flushes; LP uses the same barrier flush heuristic as LH. We also do not address the issue of using programmer annotations to prefetch data.

LP's annotations consist of an extra parameter to `Tmk_lock_acquire` calls that specifies the address of data guarded by the lock. All diffs corresponding to this `lock_pg` are either appended to the lock grant message, or sent in additional, unreliable messages.

LP's annotations only allow a single virtual memory page to be specified. However, we found that nearly all access misses either occur on the single `lock_pg`, or can not be predicted ahead of time because they result from linked list traversals through shared data structures. Therefore, LP's single page limitation does not seriously affect performance.

Four of the five applications used in this comparison are from the application suite of Chapter 3: IS, MIP, TSP, and Water. The applications were run with the same parameters as before. We also evaluated QuickSort (QS), a parallel integer sorting application that resorts to bubblesort when less than 1000 elements remain. Our input size was 262144 integers. Two of the programs, IS and Water, use barriers in addition to locks.

#### Performance

Table 3.4 lists running times, message counts, data counts, and access misses for eight-processor executions of the five applications under LH and LP, and LI. `Diffs_Requested` is a three-part statistic that provides a breakdown of the diffs that are requested in a run of the application. `Lock_pg` indicates the number of diffs requested for the current `lock_pg`. `Not_pg` is the number of diffs requested for pages other than `lock_pg`, and `Other` is the number of diffs requested while no locks are held. The total number of diffs requested is higher than the number of access misses because multiple diffs are often required to satisfy a single miss.

| Prog | Prot | Time (secs) | Msgs | Data (KBytes) | Access Misses | Diffs Requested | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | Lock Pg | Not Pg | Other |
| IS | LI | 1.75 | 1099 | 380 | 150 | 343 | 0 | 280 |
| | LH | 1.73 | 1401 | 523 | 14 | 28 | 0 | 21 |
| | **LP** | 1.74 | 1387 | 564 | 13 | 0 | 0 | 21 |
| MIP | LI | 23.91 | 21348 | 2115 | 3619 | 4532 | 9218 | 4641 |
| | LH | 19.09 | 20024 | 2375 | 723 | 279 | 795 | 1543 |
| | **LP** | 20.59 | 22552 | 2490 | 3257 | 22 | 9327 | 3586 |
| QS | LI | 14.24 | 11545 | 14670 | 2573 | 6334 | 0 | 3170 |
| | LH | 13.96 | 10055 | 15319 | 1454 | 45 | 0 | 2939 |
| | **LP** | 12.88 | 9134 | 10820 | 1232 | 0 | 0 | 3091 |
| TSP | LI | 44.09 | 19171 | 5766 | 8107 | 159 | 30497 | 10 |
| | LH | 42.61 | 10732 | 5932 | 3077 | 33 | 6315 | 5 |
| | **LP** | 44.21 | 19121 | 5848 | 8050 | 0 | 30605 | 8 |
| Water | LI | 10.63 | 26065 | 8887 | 3325 | 3162 | 200 | 4917 |
| | LH | 10.86 | 23920 | 9121 | 1740 | 283 | 34 | 2781 |
| | **LP** | 10.80 | 23521 | 9832 | 1565 | 3 | 203 | 2705 |

**Table 3.4** Protocol Tradeoffs

LP performs better than LH for only two applications: QS and Water. In the latter case the difference is minor, and both of the hybrid protocols perform worse than LI.

QS's access misses can be divided into two categories: control and data. *Control* misses are either misses on data specifying a portion of array on which to compute, or on data that is part of the implementation of the *pause_flags* used for synchronization. *Data* misses are misses on pages containing array data. We conclude that both LP and LH eliminate nearly all of the control misses, because of the low `lock_pg` and `not_pg` totals. However, the `other` category indicates that neither is able to significantly reduce the number of data misses. The difference in performance between the two protocols is caused by the differences in the amount of hybrid diff data sent. Overall, LH sends almost 50% more data than LP, and almost none of this data is useful in eliminating data misses because the accesses to array data have little temporal locality.

Only MIP and TSP request a large number of `Not_Pg` diffs. In both cases, the diffs are requested because of linked list traversals of shared data, and could not have been predicted in advance by a programmer. Therefore, LP's performance would probably not improve even if more than a single `lock_pg` were specified. LH eliminates the majority of these diff requests.

Three of the applications request a large number of `Other` diffs. Although the cause of these requests is not totally clear, many of them are certainly caused by traversals of complicated shared data structures that could not have been predicted in advance. The exception is Water, in which computation during certain phases is protected by barriers.

Overall, the results suggest that LH's heuristic is very effective in reducing misses, but occasionally at the expense of sending far more data than necessary.

## 3.4 Performance Prediction

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

### 3.4.1 Simulation Methodology

Our primary concern in selecting a simulation methodology was the ability to accurately model 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` [BL92], a profiling tool that rewrites an executable program to incorporate instrumentation code that produces an estimated processor cycle count. To account for the time spent in the operating system handling page faults and passing messages, for example, we link the program to a library that intercepts system calls and adds a specified number of cycles to the process'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 processes, the library piggybacks a process's cycle count on its synchronization messages, and adjusts the synchronizing processes' clocks according to the following rules. Processes acquiring locks must have a cycle count greater than when the lock was released by the lock's last owner. Processes departing from a barrier must have cycle counts greater than the highest cycle count among the processes arriving at the barrier.

We simulate a switched network similar to an ATM LAN. We account for contention for each point-to-point link by simulating the serialization of messages requiring access to the same link, but we do not model contention for switch resources.

To validate the simulator, we compared our model's simulated speedups to actual speedups 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.

All results in this section are for eight-processor executions.

### 3.4.2 Effect of Communication Software Speed

The results of Section 3.3.2 suggest that reducing the cost of the communication software should improve performance. This cost has two components: a fixed per message cost, regardless of message size, and a per byte cost that reflects message copying in the communication software. Figures 3.13 to 3.20 show the simulated performance of an ATM network while varying the fixed cost software overhead in the 8 processor case. All the applications that do not already achieve near-linear speedup dramatically improve as the fixed per message cost drops to zero. The large speedups indicate the performance potential for the protocols, and the potential gains to be had from hardware support for message passing.

With a low fixed cost per message, there is no longer a significant communication penalty on a switched network, reducing the impact of access misses on performance. The eager protocol gains the most by the reduction in fixed cost overhead because it uniformly suffers more access faults than the lazy protocols and therefore usually uses more messages. However, in no case did reducing the per message cost allow EI's performance to overtake the performance of the lazy protocols. LH loses ground to both of the invalidate protocols as the per message cost drops because the hybrid's advantage in the number of access misses becomes less important.

Figures 3.21 to 3.28 present the effect on speedup of varying the per byte software cost. Seven cycles per byte is the cost that we derived from our current implementation. Two cycles per byte represents a single kernel copy of the message, and zero represents no copies. The most dramatic trend in these figures is again the increased relative performance of EI. EI usually sends far more data than the lazy protocols, and therefore is much more dependent on the per byte cost. EI's performance overtakes the performance of the lazy protocols for Barnes, ILINK, and TSP when the per byte penalty drops to zero.

### 3.4.3 Effect of Network Speed

Access to the communication medium is a prime candidate for a bottleneck in any distributed system. This section examines the effects of bandwidth variation on protocol performance.

Figures 3.29 to 3.36 summarize changes in speedup for the programs when we vary bandwidth per link from 10 Mbits/sec to 1 Gigabit/sec. 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, at 100 Mbits/sec, the wire time is comparable to the per byte software cost. Increasing the network bandwidth effectively shifts the bottleneck to the software. At a bandwidth of 10 Mbits/sec, LI outperforms the other protocols for nearly all of the applications. This is because it sends the lowest amount of data as compared to the other two protocols. EI's performance suffers significantly in comparison to the lazy protocols because it sends much more data.

At other network speeds the results are less uniform. At a higher bandwidth, LH usually performs the best because there is no longer a high penalty for any extra data communicated.

An exception to these generalized results is FFT (Figure 3.30). EI shows the best performance for FFT regardless of bandwidth or software costs because it does not create either diffs or twins to communicate the full page modifications.

## 3.5 Summary

This chapter had three basic findings. First, our results show that seven of the eight programs in our suite perform better on lazy protocols than on EI. Four of the programs performed at least 18% better. The performance of FFT illustrates one pitfall of LRC. FFT performs worse under LRC than under EI because LRC always ships modified data as diffs, even when entire pages are modified. When pages are entirely re-written and shared in a migratory manner, such as in FFT, EI wins by simply shipping entire pages.

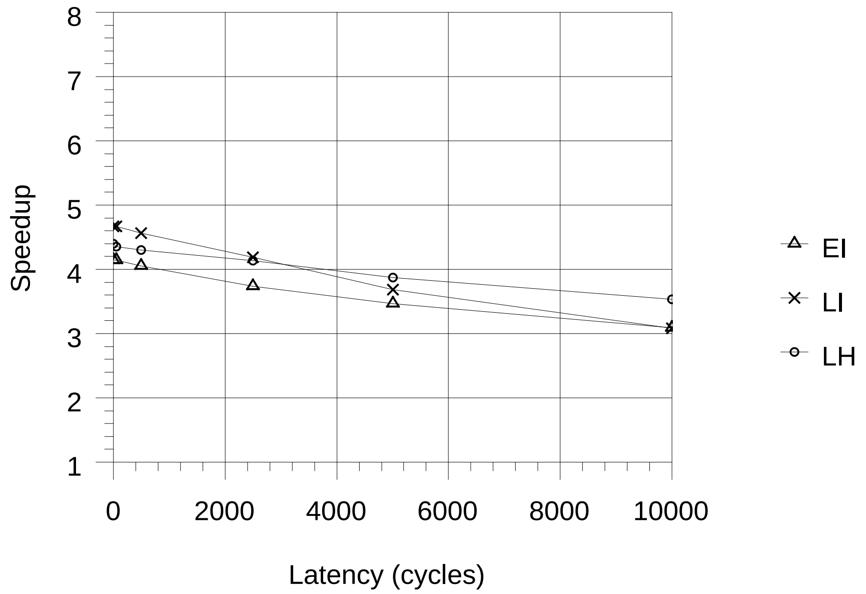


**Figure 3.13** Barnes: Varying Fixed Message Cost

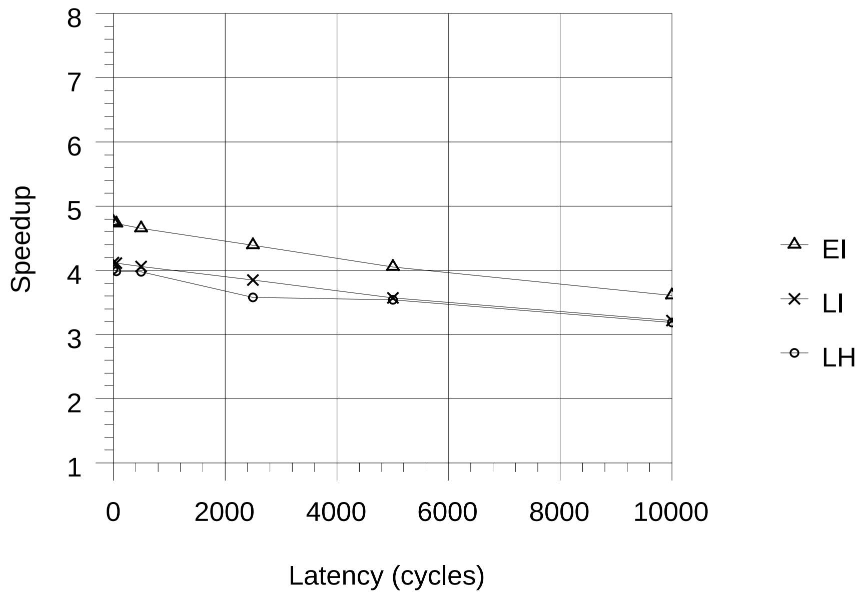


**Figure 3.14** FFT: Varying Fixed Message Cost

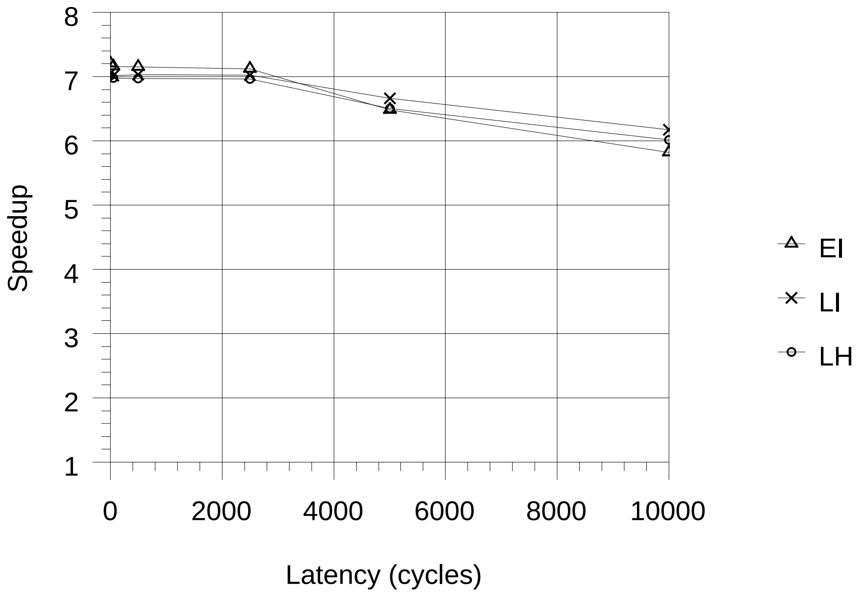


**Figure 3.15** ILINK: Varying Fixed Message Cost

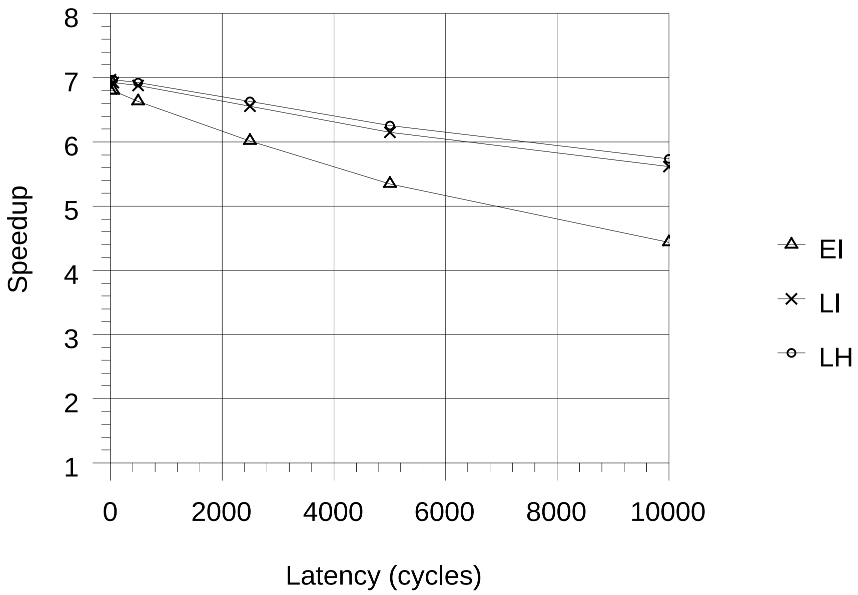


**Figure 3.16** IS: Varying Fixed Message Cost

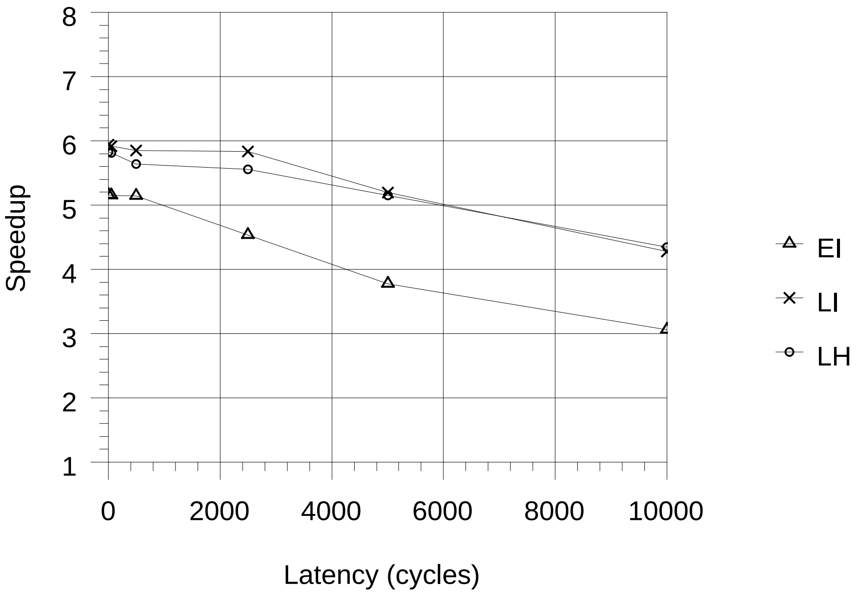


**Figure 3.17** MIP: Varying Fixed Message Cost

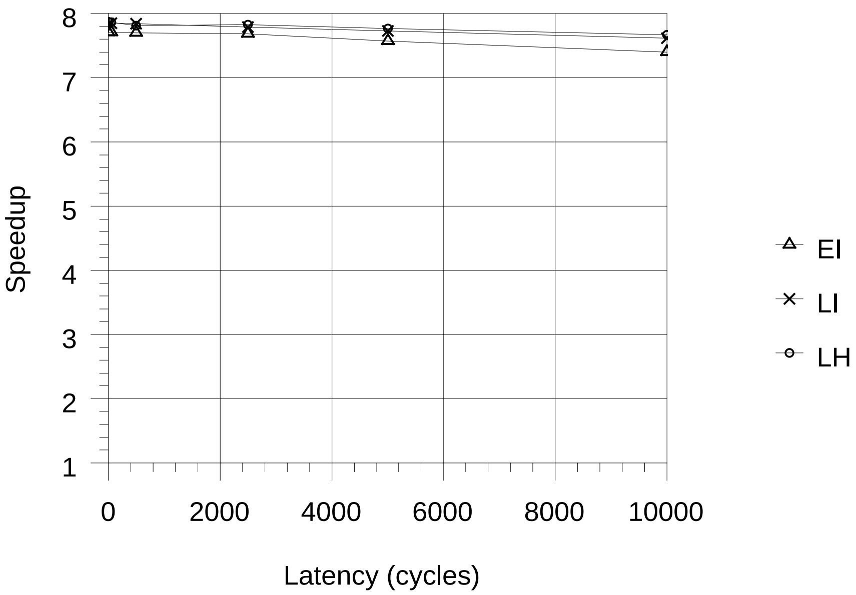


**Figure 3.18** SOR: Varying Fixed Message Cost

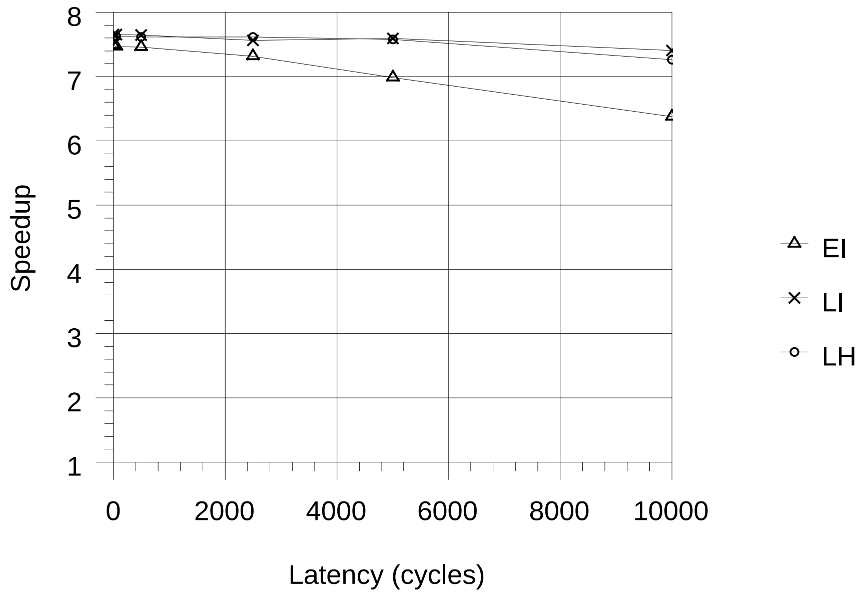


**Figure 3.19** TSP: Varying Fixed Message Cost

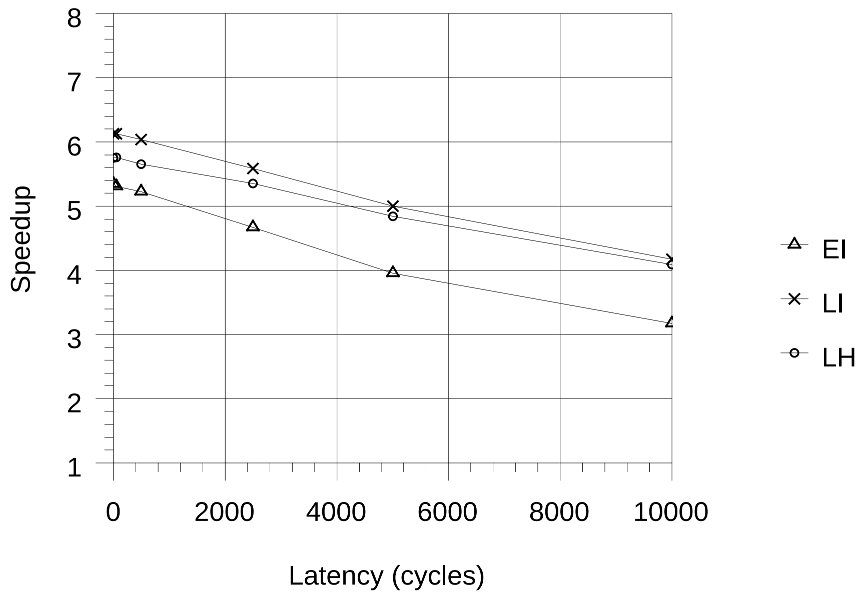


**Figure 3.20** Water: Varying Fixed Message Cost

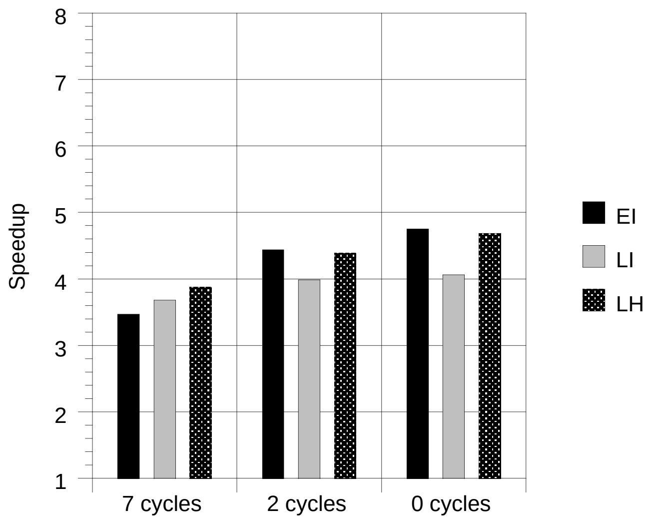


**Figure 3.21**
Barnes: Varying Per Byte Cost

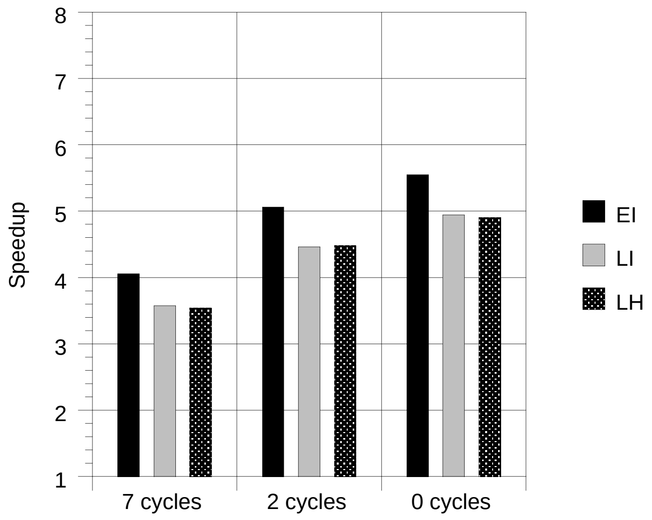


**Figure 3.22** FFT: Varying Per Byte Cost

Second, we showed that the cost of executing our protocol code was dwarfed by the cost of network communication, and especially the software overhead involved in sending and receiving messages. Directly or indirectly, software overhead accounts for almost 25% of application execution times. By comparison, the cost of executing the DSM protocols averaged only 1.2%.

Finally, we modified our implementation to simulate the effects of changing hardware and operating system overheads on the tradeoffs between the protocols. We

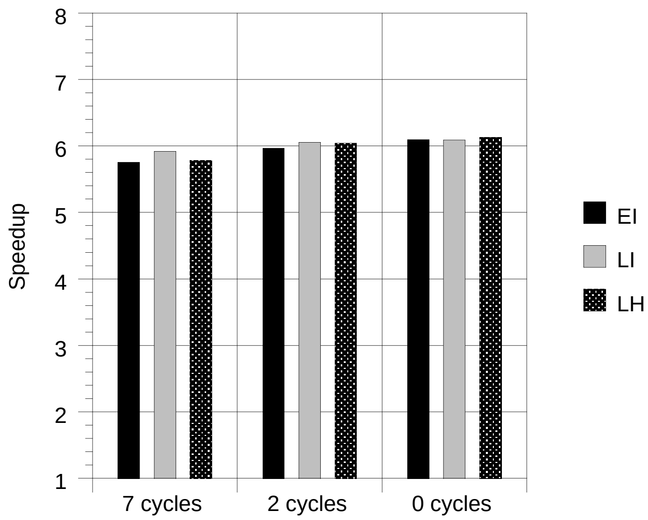


**Figure 3.23** ILINK: Varying Per Byte Cost

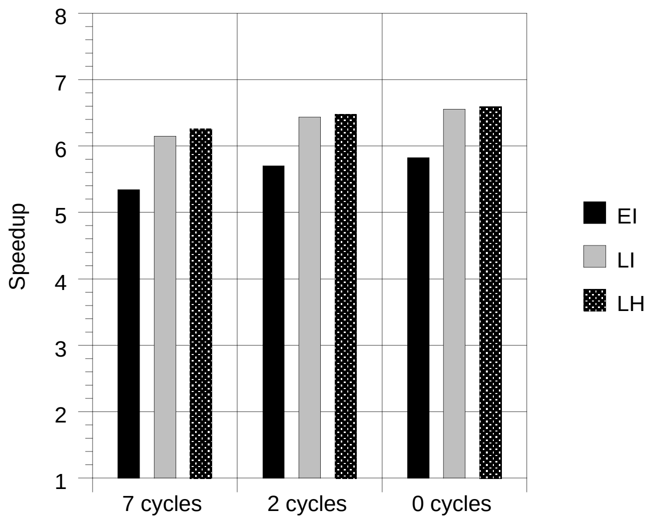


**Figure 3.24** IS: Varying Per Byte Cost

found that EI's performance improves relative to the lazy protocols for all programs as hardware and operating system overheads drop. Four of the eight programs perform better under EI than either LI or LH when the per byte cost is eliminated.

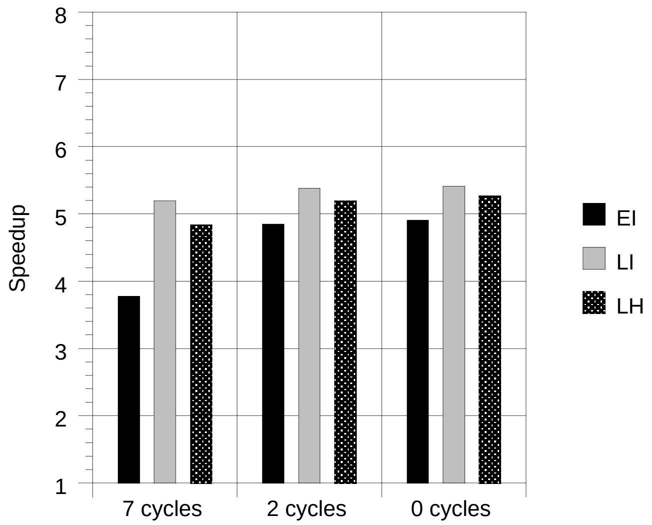


**Figure 3.25** MIP: Varying Per Byte Cost

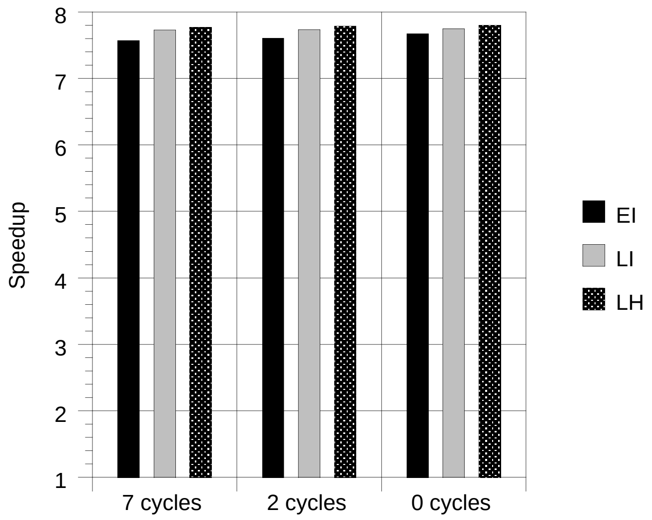


**Figure 3.26** SOR: Varying Per Byte Cost

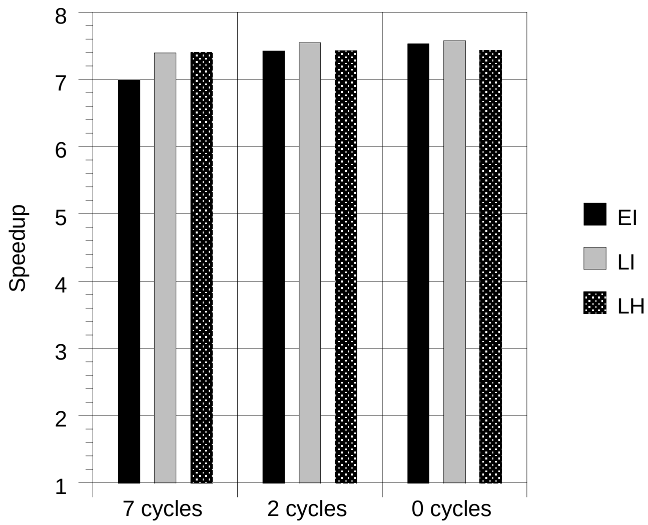


**Figure 3.27** TSP: Varying Per Byte Cost

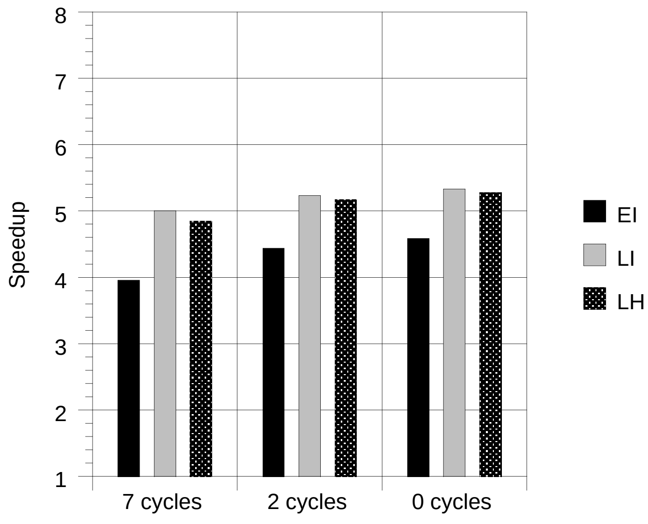


**Figure 3.28** Water: Varying Per Byte Cost

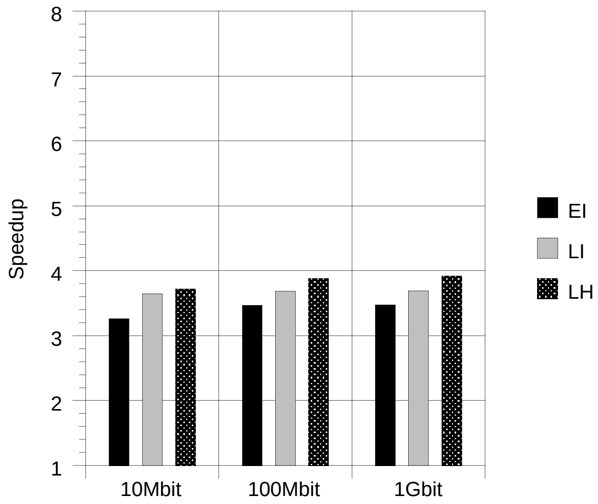


**Figure 3.29** Barnes Hut: Varying Bandwidth

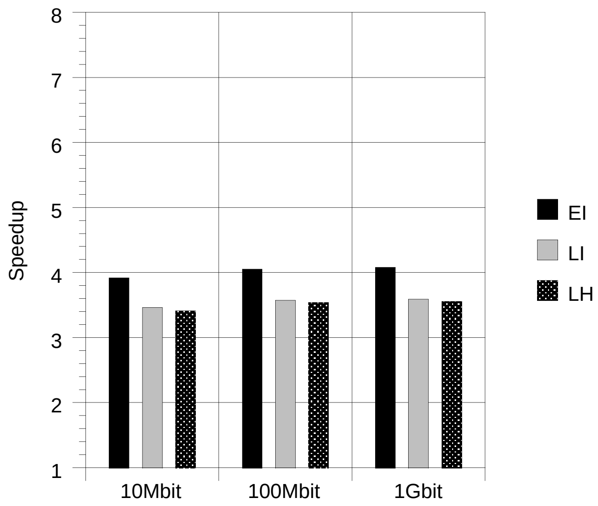


**Figure 3.30** FFT: Varying Bandwidth

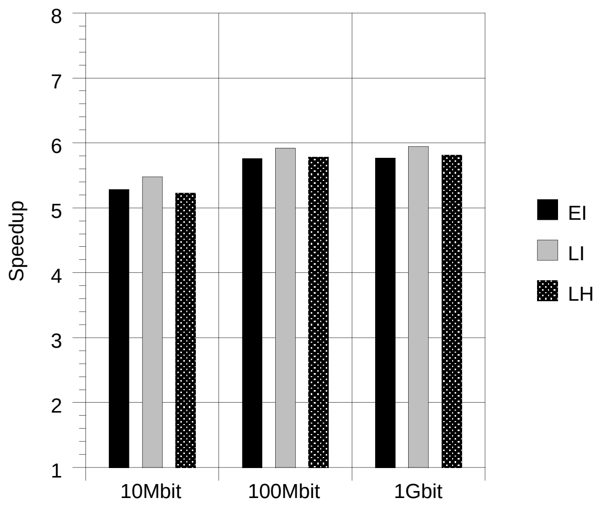


**Figure 3.31** ILINK: Varying Bandwidth

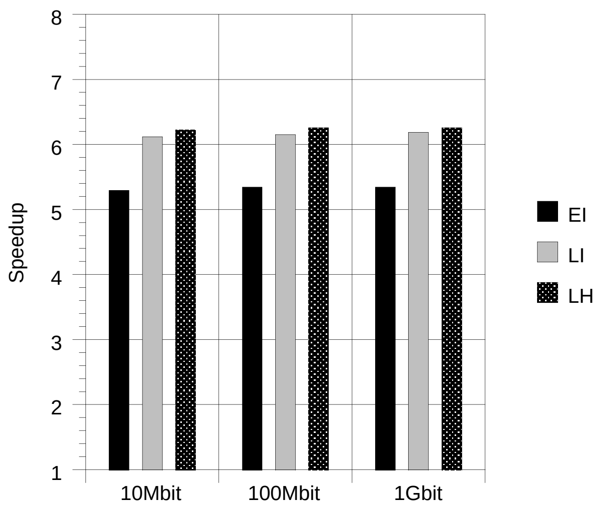


**Figure 3.32** IS: Varying Bandwidth

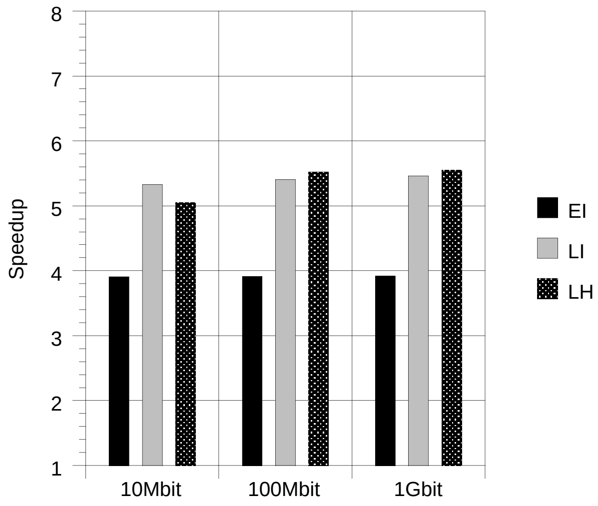


**Figure 3.33** MIP: Varying Bandwidth

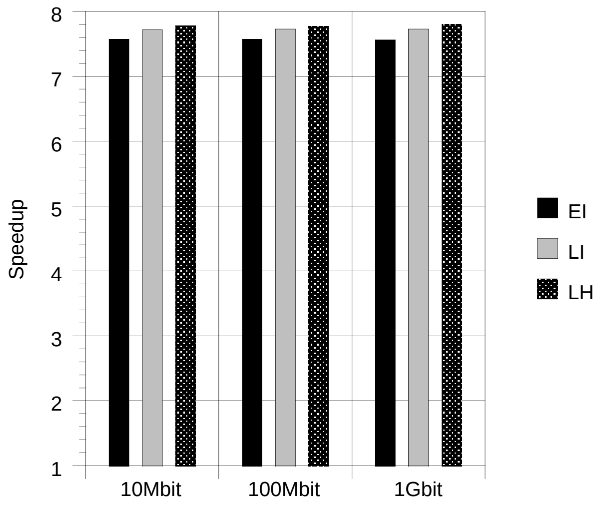


**Figure 3.34** SOR: Varying Bandwidth

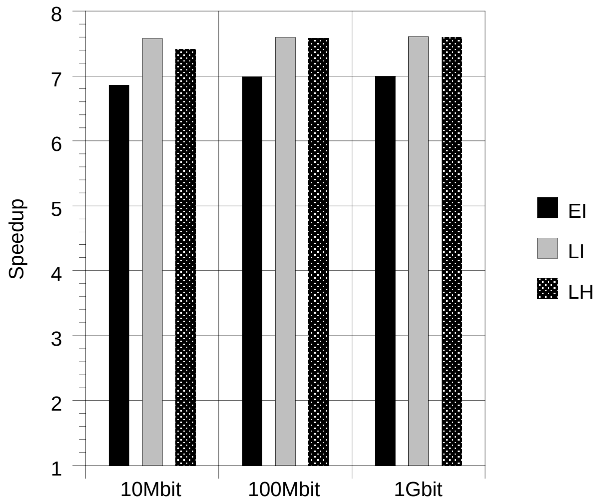


**Figure 3.35** TSP: Varying Bandwidth

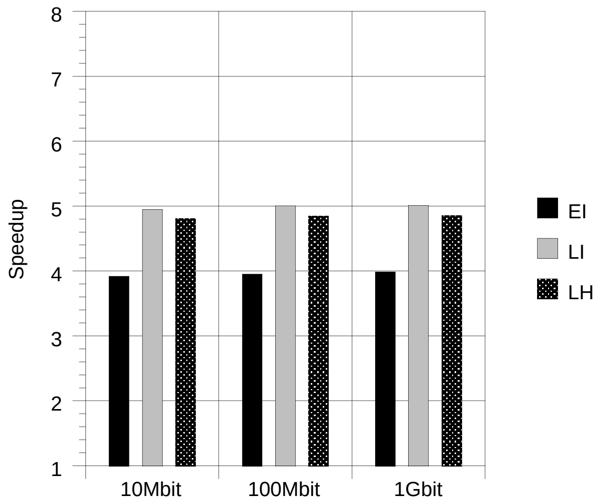


**Figure 3.36** Water: Varying Bandwidth

# Chapter 4

# Software versus Hardware

This chapter presents a detailed comparison of software and hardware approaches for supporting the abstraction of shared memory.

We first compare the performance of TreadMarks to that of an SGI 4D/480, a machine that uses the same processors and primary caches as the workstations that run TreadMarks. We show that for some classes of applications, the software approach can produce comparable or even slightly better performance than a hardware-based shared memory system. However, applications that synchronize at a fine granularity degrade more rapidly on a software system.

We then used a validated simulator to extend the comparison to larger numbers of processors, and to investigate a compromise between the hardware and software approaches. Our simulations show that for all but one application, the hardware-software approach closely tracks the hardware-only model. Again, however, applications that synchronize at too fine a granularity will only be able to achieve high performance on a hardware-based system.

## 4.1 Performance

### 4.1.1 Experimental Platforms

The system used to evaluate TreadMarks is the same as in Chapter 3. It consists of 8 DECstation-5000/240 workstations, each with a 40MHz MIPS R3000 processor, a 64 Kbyte primary instruction cache, a 64 Kbyte primary data cache, and 16 MBytes of memory. The data cache is write-through with a write buffer connecting it to main memory. The workstations are connected to a high-speed ATM network using a Fore Systems TCA-100 network adapter card supporting communication at 100 Mbits/second. In practice, however, user-to-user bandwidth is limited to 30 Mbits/second. The ATM interface connects point-to-point to a Fore Systems ASX-100 ATM switch, providing a high aggregate bandwidth because of the capability for simultaneous, full-speed communication between disjoint workstation pairs.

The shared-memory multiprocessor used in the comparison is a Silicon Graphics 4D/480 with 8 40MHz MIPS R3000 processors. Each processor has a 64 Kbyte primary instruction cache and a 64 Kbyte primary data cache. The primary data cache implements a write-through policy to a write buffer. In addition, each processor has a 1 MByte secondary cache implementing a write back policy. The secondary caches and the main memory (128 MBytes) are connected via a 16 MHz 64-bit wide shared bus. Cache coherence between the secondary caches is maintained using the Illinois protocol. The presence of the write buffer between the primary and the secondary cache makes the memory processor consistent. The SGI runs the IRIX Release 4.0.1 System V operating system.

An important aspect of our evaluation is that the DECstation-5000/240 and the SGI 4D/480 have the same type of processor running at the same clock speed, the same size primary instruction and data caches, and a write buffer from the primary cache to the next level in the memory hierarchy (main memory on the DECstation, the secondary cache on the SGI). For both machines, we use the same compiler, gcc 2.3.3 with -O optimization, and the program sources are identical (using the PARMACS macros). The only significant difference between the two parallel computers is the method used to implement shared memory: dedicated hardware versus software on message-passing hardware.

Single processor performance on the two machines depends on the size of the program's working set. Both machines are the same speed when executing entirely in the primary cache. If the working set fits in the secondary cache on the 4D/480, a single 4D/480 processor is 2% to 3% slower than a DECstation-5000/240 because the main memory of the DECstation-5000/240 is slightly faster than the secondary cache of the 4D/480 processor. (The 4D/480's secondary cache is clocked at the same speed as the backplane bus, 16 MHz.) If the working set is larger than the secondary cache size, the 4D/480 slows down significantly.

### 4.1.2 Application Suite

The application suite used for our hardware versus software comparisons differs from that of Chapter 3 because we no longer have access to the SGI, and several of the applications in our current suite were not yet available when we lost access. Nonetheless, the applications provide a reasonably broad spectrum of parallel programs.

We present results from our modified version of Water, the original Water (hereafter referred to as O-Water), SOR, TSP, and ILINK.

We ran O-Water and Water on 288 molecules for 5 time steps.

We ran SOR on a $2000 \times 1000$ and a $1000 \times 1000$ matrix. We chose the $2000 \times 1000$ problem size because it does not cause paging on a single DECstation, and it fits within the secondary cache of the 4D/480 when running on 8 processors. The $1000 \times 1000$ run is included to assess the effect of changing the communication to computation ratio.

Both 18- and 19-city problems were used as input to TSP.

We ran ILINK with two different inputs, CLP [HWC+93] and BAD [LRC+92], both corresponding to real data sets used in disease gene location.

### 4.1.3 Results

Table 4.1 presents the single processor execution times on both systems, as well as the single processor DECstation time without TreadMarks. As can be seen from this table, the presence of TreadMarks has almost no effect on single processor execution times. However, the SGI can be significantly slower for programs that do not fit into the secondary cache.

Figures 4.1 presents the speedups achieved for ILINK, SOR, TSP, O-Water and Water, both on TreadMarks and the 4D/480. The TreadMarks speedups are relative to the single processor DECstation run times without TreadMarks.

| Program | DEC | TreadMarks | SGI |
|---|---|---|---|
| ILINK-CLP | 6352.4 | 6388.0 | 6208.0 |
| ILINK-BAD | 858.1 | 860.4 | 936.1 |
| SOR $2000 \times 1000$ | 416.9 | 419.6 | 581.6 |
| SOR $1000 \times 1000$ | 229.5 | 230.3 | 315.1 |
| TSP-19 | 308.6 | 310.3 | 318.8 |
| TSP-18 | 25.4 | 25.5 | 26.3 |
| O-Water-288-5 | 43.1 | 44.4 | 44.4 |
| Water-288-5 | 43.1 | 43.7 | 44.1 |

**Table 4.1** TreadMarks vs. SGI 4D/480

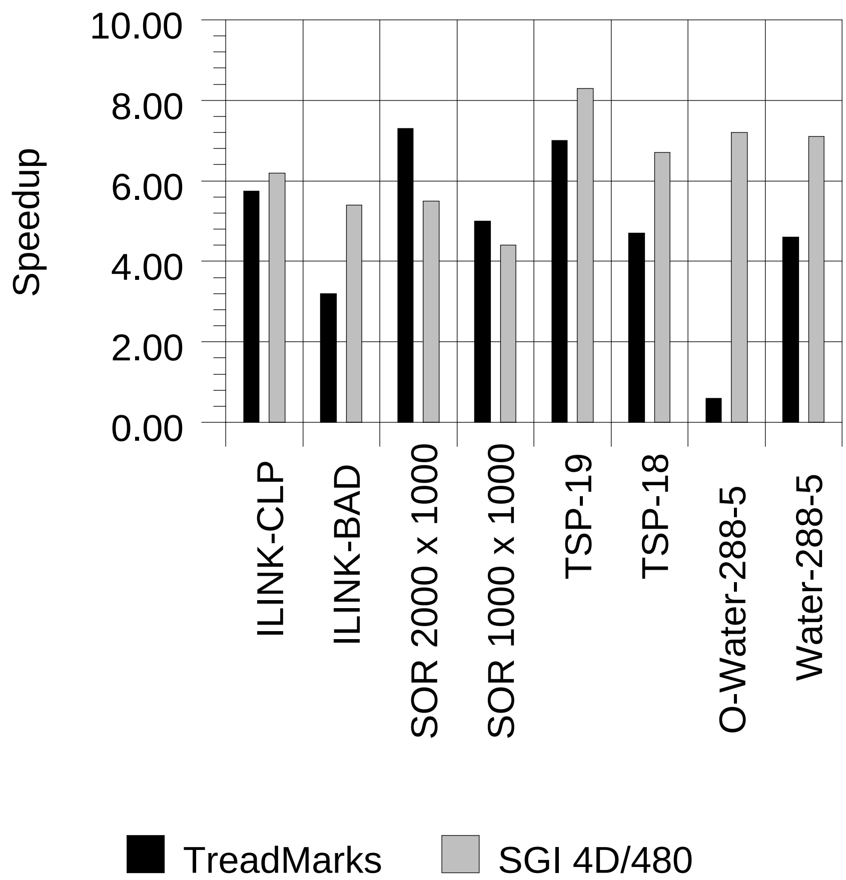


**Figure 4.1** 8-Processor Speedup: TreadMarks vs. SGI 4D/480

**ILINK**

We ran ILINK with two different inputs, CLP and BAD. These inputs show the best and the worst speedups, respectively, among the inputs that are available to us[§]. CLP exhibits the smallest difference in speedup between TreadMarks and the 4D/480, 5.74 vs. 6.24, and BAD exhibits one of the largest differences, 3.15 vs. 5.41.

[§]We have since received two more inputs sets that perform better than CLP and are more representative of linkage analysis in general.

ILINK achieves less than linear speedup on both the 4D/480 and TreadMarks because of a combination of load balancing problems and the cost of several sections of code that are implemented sequentially rather than in parallel [DSC+94].

The 4D/480 outperforms TreadMarks because of the large amount of communication. The communication rate for the CLP input set is 157 Kbytes/second and 449 messages/second on 8 processors, compared to 526 Kbytes/second and 1,800 messages/second for the BAD input set, hence the better speedups achieved for CLP.

### SOR

We ran our Red-Black SOR with two different problem sizes: $2000 \times 1000$ and $1000 \times 1000$. Of the four applications used, SOR is the only one for which there is a sizable difference in single processor execution time between TreadMarks and the 4D/480. TreadMarks is approximately 25% faster on a single processor, because both problem sizes exceed the size of the secondary cache on the SGI.

In addition to lower single processor execution times, better speedups are achieved on TreadMarks. The difference is partly due to the way in which TreadMarks communicates updates to shared memory. Points at the edge of the matrix are initialized to values that remain fixed throughout the computation. Points in the interior of the matrix default to 0. During the early iterations, the points at the interior of the array are recomputed (and stored to memory) but their value remains the same. Only the points near the edge change value. On the 4D/480, the hardware cache coherence protocol updates the memory regardless of the fact that the values remain the same. TreadMarks, however, only communicates the points that have changed value because *diff*s (see Chapter 2) are computed from the contents of a page. Consequently, the amount of data movement by TreadMarks is significantly less than the amount of data movement by the 4D/480. The estimated data movement by the 4D/480 after the initial data migration between processors is 5567 Kbytes, whereas the actual data movement by TreadMarks is 1045 Kbytes.

To eliminate this effect, we initialized the matrix such that every point changes value at every iteration, equalizing the data movement by the 4D/480 and TreadMarks. Even in this modified version, the speedup is still better on TreadMarks than on the 4D/480. We attribute this result to the fact that most communication in SOR occurs at the barriers and between neighbors. This communication occurs in parallel on the

ATM network. On the 4D/480, the bus interferes with the primary cache's access to the secondary cache because the secondary cache does not have dual tags.

### TSP

We solved both 18-city and 19-city Traveling Salesman Problems. Branch-and-bound algorithms can achieve super-linear speedup if the parallel version finds a good approximation early on, allowing it to prune more of the search tree than the sequential version. An example of such super-linear speedup can be seen on the 4D/480 for the 19-city problem. More important than the absolute values of the speedups is the comparison between the speedups achieved on the two systems. We see better performance on the 4D/480 than on TreadMarks (8.35 vs. 7.02 for the 19-city problem and 6.67 vs. 4.71 for the 18-city problem). The difference is slightly larger for the 18-city problem because of the increased synchronization and communication rates.

Again, the performance of TSP on TreadMarks suffers from the fact that TSP is not *data-race-free* (see Section 3.2). Although updates to the current minimum tour length are synchronized, read accesses are not. Since TreadMarks updates cached values only on an *acquire*, a processor may read an old value of the current minimum. The execution remains correct, but the work performed by the processor may be redundant because a better tour has already been found elsewhere. On the 4D/480, this is unlikely to occur since the cache consistency mechanism invalidates cached copies of the minimum when it is updated. By propagating the bound earlier, the 4D/480 reduces the amount of work each processor performs, leading to a better speedup. Adding synchronization around the read accesses would hurt performance, given the large number of such accesses.

To eliminate this effect, we modified TSP to perform an *eager* lock release instead of a lazy lock release after updating the lower bound value. With an eager release, the modified values are updated at the release, rather than at a subsequent acquire. The speedup of TSP improved from 7.02 to 7.41 on 8 processors, vs. 8.35 on the 4D/480. The remaining differences between the DSM and the SGI performance can be explained by faster lock acquisition on the SGI, compounded with the non-deterministic effect of picking up redundant work due to the slight delay in propagating the bound.

**Water**

TreadMarks gets no speedup on O-Water, except on 2 processors, because of the extremely large number of messages caused by the high synchronization rate (1,540 remote lock acquires/second).

There is a marked improvement with Water. On the 4D/480, Water's speedup is virtually identical to O-Water. On TreadMarks, however, the speedup improves to 4.61 on 8 processors.

Part of the high cost of message transmission is due to the user-level implementation of TreadMarks, in particular, the need to trap into the kernel to send and receive messages. We have implemented TreadMarks inside the Ultrix kernel in order to assess the trade-tradeoffs between a user-level and a kernel-level implementation. In comparison, the minimum time to acquire a lock drops from 0.78 to 0.43 milliseconds, and the time for an 8-processor barrier drops from 2.20 to 0.74 milliseconds. For ILINK, SOR and TSP, the differences between the kernel and user level implementations are minimal, reflecting the low communication rates in these applications. For Water, however, the differences are substantial. Speedup on 8 processors increases from 3.96 for the user-level implementation to 5.60 for the kernel-level implementation, compared to 7.17 for the 4D/480.

## 4.2 Simulation

In this section, we compare the performance of our all-software DSM (AS) to an all-hardware shared memory architecture (AH), and to one that uses both hardware and software at different levels (HS). While the DSM scales to a larger number of processors without modification, hardware architectures quickly become more complex once the number of processors exceeds the capacity of a single bus. In the case of our hardware architecture, the processor interconnect is a crossbar with one at each node, and the cache controllers implement a directory-based cache coherence protocol.

Our HS architecture consists of a number of bus-based multiprocessors, each with sufficient bus bandwidth to support the processors without contention causing a bottleneck. Conventional bus snooping hardware enforces coherence between the processors within a node. These hardware shared-memory multiprocessors then become nodes on a general-purpose network, with coherence between different nodes implemented in software. We will refer to these three architectures as the All Software (AS), All Hardware (AH), and Hardware-Software (HS) approaches.

The HS approach is promising both in terms of cost and in complexity. Compared to the AS approach, bus-based multiprocessors with a small number of processors ($N$) are cheaper than $N$ comparable uniprocessor workstations. Furthermore, the cost of the interconnection hardware is reduced by roughly a factor of $N$. Compared to the AH approach, commodity parts can be used, reducing the cost and complexity of the design. In this section, we assess the performance of the HS approach compared to AS and AH.

### 4.2.1 Simulation Models

We modeled the architectures and simulated the programs using an execution-driven simulator [CDJ+91]. Instead of the DECstation-5000/240 and SGI 4D/480, we base our models on leading-edge technology. All of the architectural models use RISC processors with a 150 MHz clock, 64 Kbyte direct-mapped caches with a block size of 32 bytes, and main memory sufficient to hold the simulated problem without paging. We simulate up to 64 processors for each architecture.

In both the AH and the AS models, each node has one processor and a local memory module. A cache miss satisfied by local memory takes 12 processor cycles. In the HS model, each node has 8 processors connected by a 256-bit wide split transaction bus operating at 50 MHz. A cache miss satisfied by local memory takes 16 to 18 processor cycles, which is slightly longer than the AH and the AS models because of bus overhead.

In the AH model, the nodes are connected by a crossbar network with point-to-point bandwidth of 200 MBytes/second and a latency of 160 nanoseconds. We used a crossbar in order to minimize the effect of network contention on our results. The point-to-point bandwidth is the same as the Intel Paragon's network. Cache coherence is maintained using a directory-based protocol. A cache miss satisfied by remote memory takes 92 to 130 processor cycles, depending on the block's location and whether it is modified. These cycle counts are similar to the Stanford DASH [LLG+92] and FLASH [Kea94] multiprocessors.

In both the AS and the HS models, the general-purpose network is an ATM switch with a point-to-point bandwidth of 622 MBit/second and a switching latency of 1 $\mu$sec. Memory consistency between the nodes is maintained using the TreadMarks LRC invalidate protocol (See Chapter 2.3.1). In addition, the simulations account for the wire time, contention for the network links, and the software overhead of entering the kernel to send or receive messages, including data copying ($5000+28\times$ *message size in words* processor cycles), calling a user-level handler for page faults and incoming messages (4000 processor cycles), and creating a diff ($8\times$ *words per page* processor cycles). The values are based on measurements of the TreadMarks implementation on the DECstation-5000/240 (See Chapter 2).

For the HS approach, all of the processors within a node are treated as one by the DSM system. We assume that cache and TLB coherency mechanisms will ensure that processors within a node see up-to-date values. Multiple faults to the same page are merged by the DSM system. In other words, if one processor faults on a page and later another processor faults on the same page, the second and subsequent processors simply wait until the first processor has retrieved the page. Synchronization is implemented through a combination of shared memory and message passing, reflecting the hierarchical structure of the machine. For barriers, each processor updates a local counter until the last processor on the node has reached the barrier. The last processor sends the arrival message to the manager. When the last arrival message arrives at the manager, it issues a departure message to each node. Similarly, locks are implemented using a *token*. The token is held at one node at a time. In order to acquire a lock, a processor must first bring the token to its node. If the token already resides at the node, no messages are required.

### 4.2.2 Validation

We compared our model's simulated speedups to actual speedups for 4 and 8 processors on the applications O-Water, SOR, and TSP. The results are summarized in Table 4.2. In all cases, simulated speedup, the number of messages, and the total

| Program | Type | 4 | 8 | 16 |
|---|---|---|---|---|
| Water<br>343 Molecules | Sim | 3.06 | 4.84 | 5.88 |
| | Real | 2.76 | 4.54 | |
| SOR<br>Red-Black | Sim | 3.89 | 7.46 | 13.69 |
| | Real | 3.71 | 7.33 | |
| TSP<br>19 Cities | Sim | 3.97 | 7.30 | 12.50 |
| | Real | 3.90 | 7.08 | |

**Table 4.2** Real and Simulated Speedups

amount of data communicated came to within 10% of the actual numbers.

### 4.2.3 Results

We simulated SOR, TSP, and Water. Excessively long simulation times prevented us from including simulation results for ILINK. Water was run with 288 molecules, TSP for 19 cities, and SOR for a 4000 by 2000 element array. Figures 4.2 to 4.4 report the speedups achieved on the three different architectures. Since the uniprocessor execution times are roughly identical for all three architectures, the execution times are omitted. Figures 4.5 and 4.6 present the message and data movement totals for AS and AH relative to the AS numbers. Finally, the rest of the section discusses the observed performance of the individual applications.

#### SOR

Figure 4.2 presents speedups for the SOR program for a $2000 \times 1000$ matrix. Since we only simulate a small number of iterations, we begin the simulation with the second iteration in order to prevent cold start misses from dominating our statistics. Linear speedup is achieved on AH and HS, while the performance of AS is sub-linear due to the high communication cost. SOR performs mainly nearest neighbor communication. Hence this program can take advantage of the hierarchical nature of the HS architecture. The only processors to incur a high penalty for misses are the edge processors that share data with processors that are off-node, and hence this program incurs little extra overhead on HS in comparison to AH. This conclusion is supported by the observation that the number of messages for the 64-processor execution on HS is 1/9 of the number of messages for the 64-processor AS execution (See Figure 4.5).

#### TSP

Figure 4.3 presents speedups for the TSP program with a 19 city input. This program has a very high computation to communication ratio. However, as the number of processors increases, this ratio decreases enough for the high latency of communication in the AS architecture to become a bottleneck. Figure 4.5 shows that the number of messages for the HS architecture is less than 1/2 that for the AS architecture. The reduction is not 8-fold because the next processor to access the queue is more likely to be from another node. Figure 4.6 shows that the amount of data movement by HS

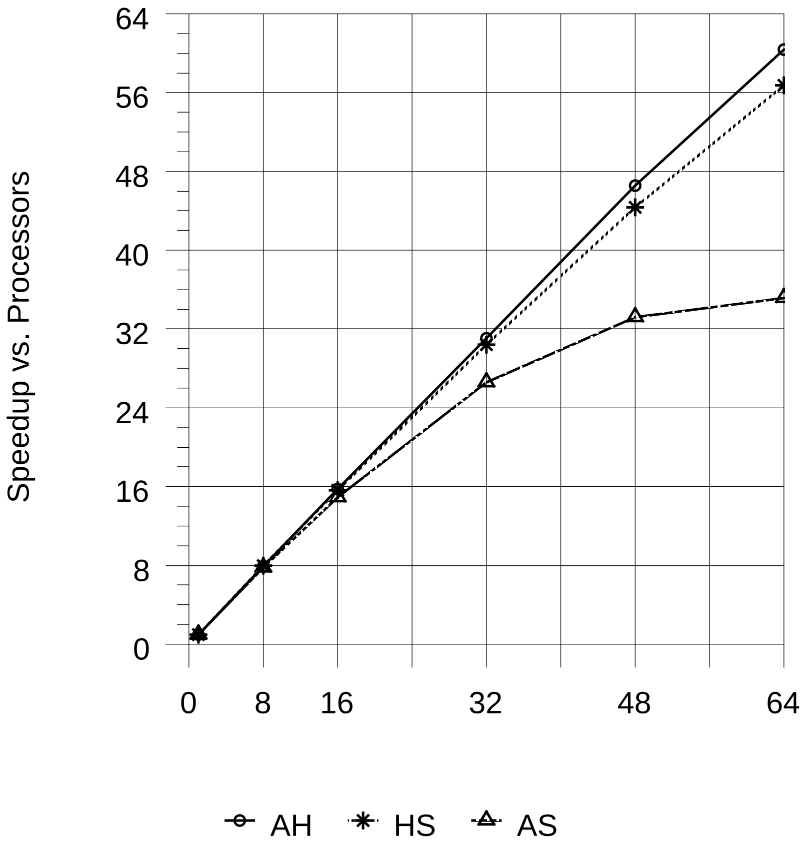


**Figure 4.2** Speedups for SOR: $2000 \times 1000$

is about 1/8 that for AS. The 8-fold reduction in data movement is a result of HS coalescing changes from different processors on a node into a single diff.

**Water**

Figure 4.4 presents speedups for Water running 2 time steps on 288 molecules. Beyond 32 processors, AH is the only architecture whose speedup improves. AS obtains a peak speedup of $X$ at 16 processors, and HS reaches its peak speedup of $Y$ at 32 processors. The performance is poor for the AS architecture because of the large number of synchronization operations as well as the large amount of data communicated. Although HS gets a 5-fold decrease in the number of *overall* messages and a 13-fold decrease

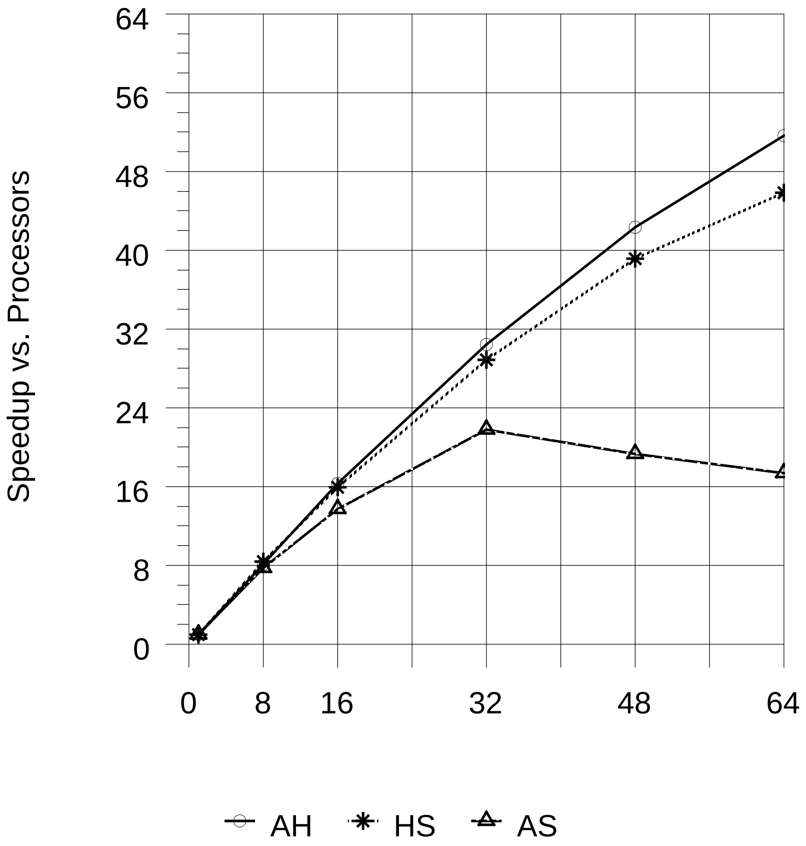


**Figure 4.3** Speedups for TSP: 19 Cities

in the amount of data movement compared to the AS architecture, its performance does not match AH because the number of synchronization messages (and the wait time to acquire the locks) remains high (See Figure 4.5).

### 4.2.4 Reduced Software Overhead

Message-passing systems with lower software overhead than Unix sockets are possible, either through optimizing the software structure, e.g., Peregrine [JZ93], or a user-level hardware interface, e.g., SHRIMP [BLA+93]. In this section, we examine the effect of reducing both the *fixed* and *per word* overheads. Specifically, we examine the effect of reducing the fixed cost from 5000 processor cycles to 500, roughly Peregrine, and

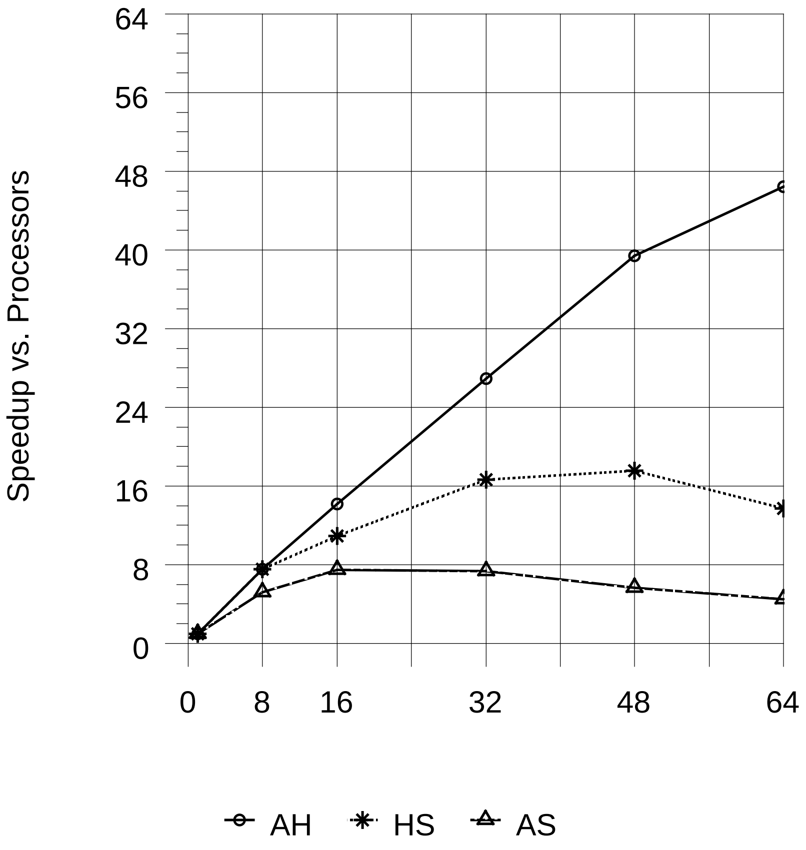


**Figure 4.4** Speedups for Water: 288 Molecules and 2 Steps

50, roughly SHRIMP, and the per word cost from 28 processor cycles to 8, one `bcopy` to the interface.

Figures 4.7 and 4.8 present the speedups for SOR and Water on the AS architecture. These show the smallest and the largest effects for reducing the software overhead. For SOR, the fixed cost has the largest effect on performance; while, for Water, both the fixed and per word cost have equal effects on performance.

Figure 4.9 presents the speedups for Water on the HS architecture. Because HS reduces the amount of data movement more than the number of messages (compared to AS), the fixed cost has a more significant effect than it did for AS.

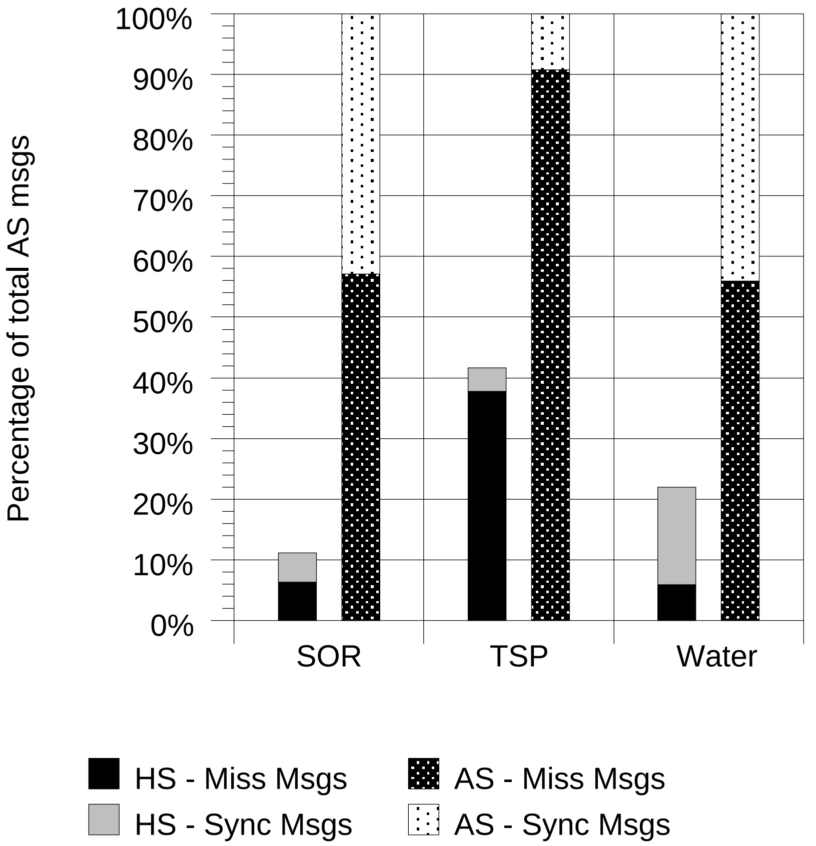


**Figure 4.5** Total Messages

## 4.3 Summary

The relative magnitude of the differences in speedup between TreadMarks and the 4D/480 for ILINK, TSP, O-Water and Water roughly correlate to the differences in the synchronization rates. For TSP, O-Water and Water, which are primarily lock based, the difference in speedup is closely related to the frequency with which off-node locks are acquired. On 8 processors, the difference in speedup is 6.7 for O-Water (with 1540 remote lock accesses per second), 3.2 for Water (680), 1.4 for the 18-city TSP (32), and 1.3 for the 19-city TSP (14). In addition, for TSP, the 4D/480 performs better because the eager nature of the cache consistency protocol reduces the amount of redundant work performed by individual processors. For ILINK, which uses barriers,

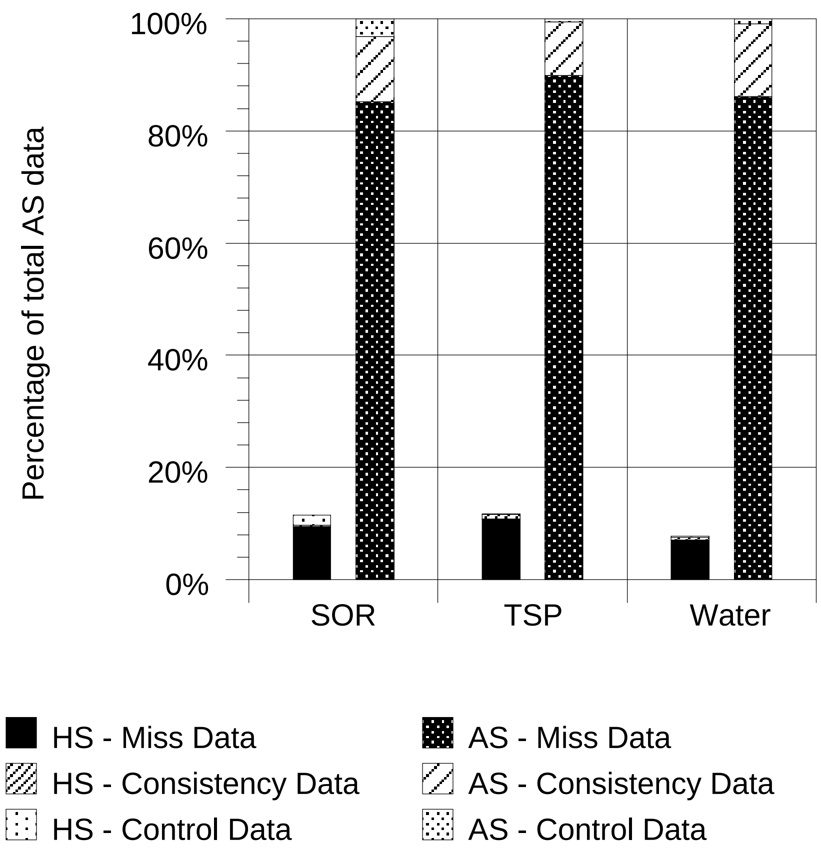


**Figure 4.6** Total Data

the difference in speedup can be explained by the barrier synchronization frequency, a difference of 2.2 for the BAD data set with 10 barriers per second, vs. a difference of 0.4 for CLP with 0.36 barriers per second. For SOR, the larger memory bandwidth available in TreadMarks results in better speedups. Dual cache tags and a faster bus, relative to the speed of the processors, are necessary to overcome the bandwidth limitation on the SGI.

The ATM LAN's longer latency makes synchronization much more expensive on TreadMarks than on the 4D/480. Moving the implementation inside the kernel, as we did, is only one of several mechanisms that can be used to reduce message latency.

Our simulation results show that the AS approach does not scale well for the applications and problem sizes that we simulated. The HS approach, which uses hardware

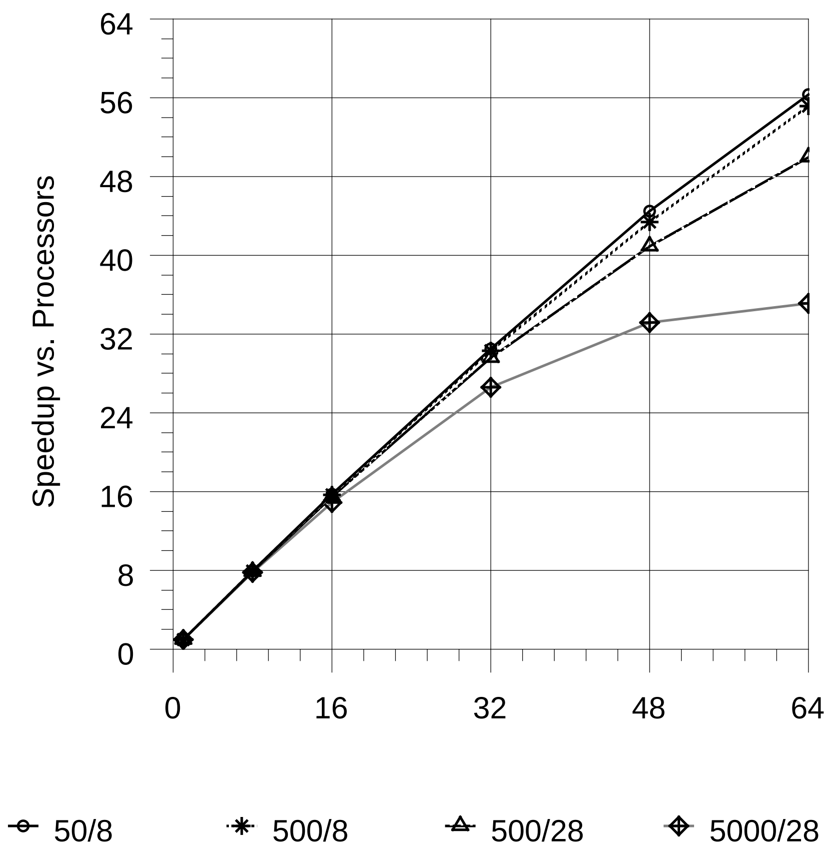


**Figure 4.7** AS Speedups for SOR: $2000 \times 1000$

for coherence at the node level and software for inter-node coherence, scales very well for SOR and TSP. For example, SOR performs nearest-neighbor sharing which takes advantage of the HS architecture, and TSP takes advantage of the coalescing of diffs. For SOR and TSP, the HS performance is almost identical to the AH approach. For Water, the frequent synchronization results in inferior performance for HS compared to AH.

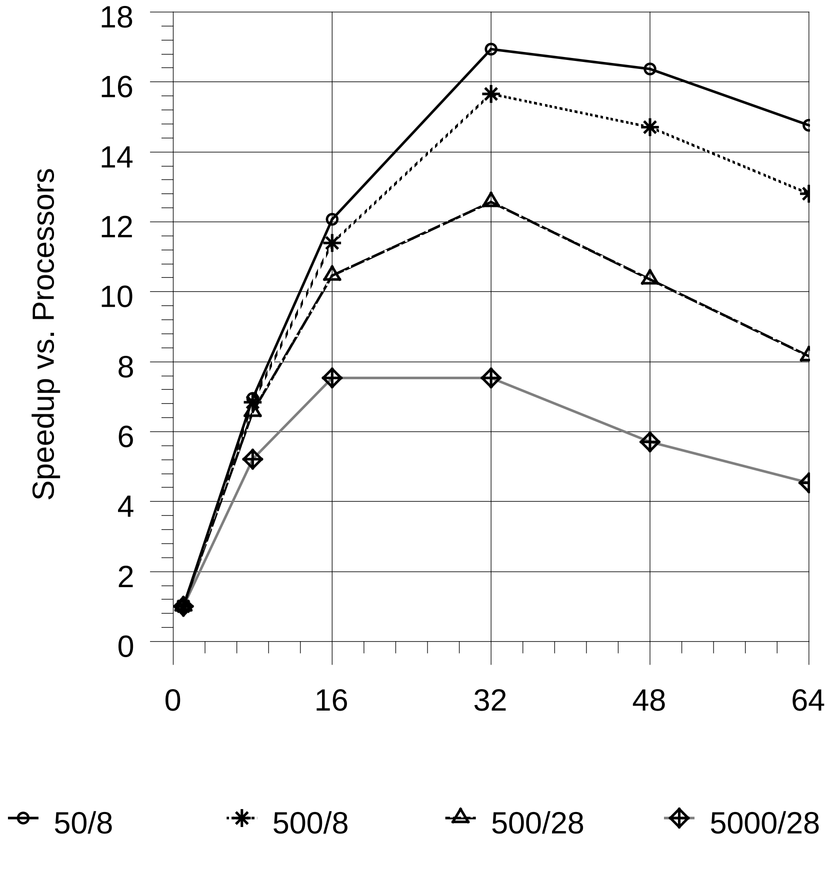


**Figure 4.8** AS Speedups for Water: 288 Molecules and 2 Steps

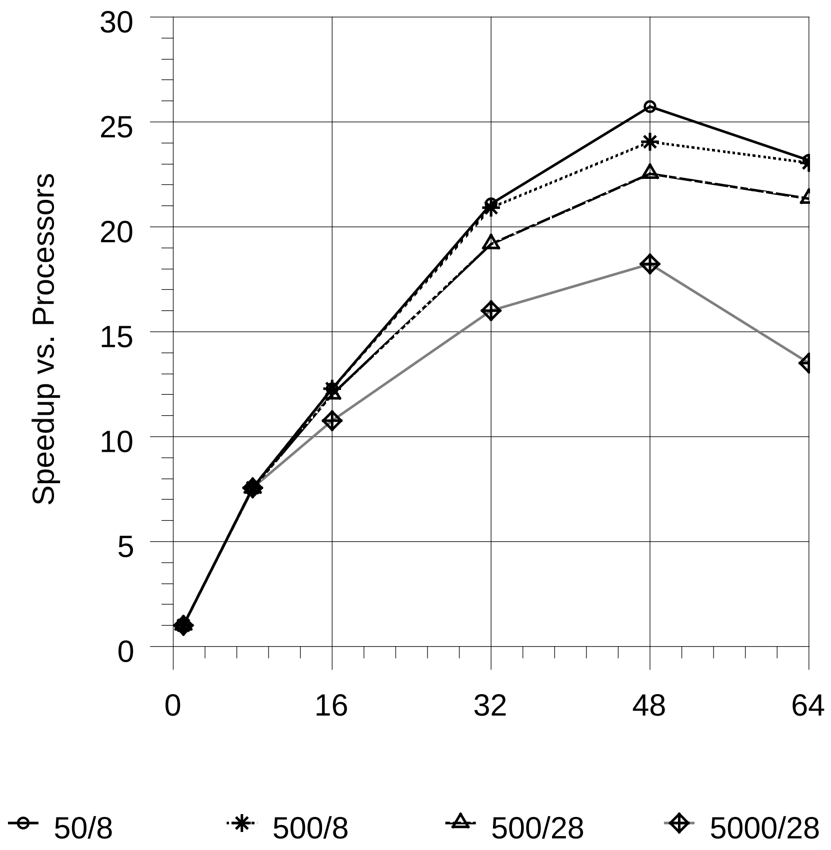


**Figure 4.9** HS Speedups for Water: 288 Molecules and 2 Steps

# Chapter 5

# Related Work

This section compares lazy release consistency and the TreadMarks system with a number of other software and hardware shared memory system, paying particular attention to differences in memory and programming models.

## 5.1 Software-Supported Shared Memory

### Sequentially Consistent Systems

Ivy [LH89] was the first software-only implementation of distributed shared memory. Ivy implemented a single-writer, sequentially consistent, invalidate protocol that used virtual memory pages as the base unit of consistency. Ivy used a dynamic, distributed page management and location scheme. Chains of "probable owners" are followed until arriving at the owner. At each intermediate site, the local "probable owner" is updated to point to the processor performing the search.

Mirage [FP89] differs from Ivy in that newly mapped pages are "pinned" to a processor for an interval of time ($\Delta$) before they can be invalidated or migrated to other processors. The duration of the pin is set at application startup and remains constant through the duration of the execution. The purpose of the pin is to prevent the "ping-pong" effect that can plague systems like Ivy.

Clouds [DCM+90] is another early sequentially consistent DSM. Clouds differs from Ivy in that it uses program-defined segments as the unit of memory granularity rather than pages, and in that it allows memory to be pinned as in Mirage. However, segments are pinned via explicit "pin" and "unpin" operations. This method creates opportunities for savvy programmers to improve application performance, but also requires them to understand more of the details of the underlying DSM system.

Mether [MF89, MF90] is another Ivy-like system that runs on top of SunOS. Mether's primary innovation is allowing both intra- and inter-program sharing. None of the other systems discussed in this chapter explicitly address inter-program sharing, but this sharing is one of the main rationales behind recent research in single address space systems [CLBHL93, SLM90].

Mether also divides shared space into regions that have different consistency mechanisms. "Demand-driven" space behaves as in a conventional system, while "data-driven" space has semantics somewhat like unix-pipes. Data must be written before it can be read and can be used to support efficient synchronization. TreadMarks handles synchronization separately from the memory system.

Bryant et al. [BCCR91] implemented Structured Shared Virtual Memory (SSVM) on a star network of IBM RS-6000s running Mach 2.5. Two different implementation strategies were followed: one using the Mach external pager interface [YTR+87], and one using the Mach exception interface [BGR+88]. They report that the latter implementation—which is very similar to ours—is more efficient, because of the inability of Mach's external pager interface to asynchronously update a page in the user's address space.

Emerald, Amber, and Orca are distributed object systems that share many similarities with DSM systems. Distributed object systems distribute computations among threads executing in objects on different machines. Processes interact by invoking methods of shared objects.

Emerald [BHJL86, BHJ+87] implements local communication among threads and objects through shared memory, and remote communication through RPC. The system marshals RPC arguments into the message, accommodating pointer values by copying data pointed to by any pointer parameters. Objects are not automatically migrated, and Emerald does not support object replication. Emerald does support programmer-directed object migration across machine boundaries.

Amber [CAL+89] eliminated Emerald's complicated RPC handling by implementing a single address space across all machines. Amber did not automatically move or replicate data, but provided special RPC variations that told the system whether to use RPC or to move the calling object to the destination site. Subsidiary objects could also be "attached" to other objects. All attached objects moved between nodes when any one of the objects moved.

Orca [BKT92, Kaa92] is an object-based system that supports transparent replication by a fast, hardware-supported, multicast mechanism. Replication is not automatic, but initiated by heuristics. Since all accesses to shared data occur through specific methods provided by the object encapsulating the data, the system easily detects modified data and does not need a mechanism like TreadMarks' diffs. While

Orca's update mechanisms are indeed efficient, they require both language and hardware support. The language support is necessary to implement and enforce the object model, while the transparent replication relies on hardware multicast support.

Munin [CBZ91] was the first DSMs to implement a relaxed consistency model. Munin was the first software release consistent system built, the first system to employ "diff"ing to detect modifications to shared data, and the first system to allow the user to employ multiple protocols to handle data with different access characteristics.

The final version of Munin distinguished between five different types of data: `conventional`, `read-only`, `synchronization`, `migratory`, and `write-shared`.

The `conventional` protocol is modeled after Ivy's distributed scheme, with the addition of Mirage's $\delta$ timeout. `Read-only` and `synchronization` data are handled as in TreadMarks. The `Migratory` protocol is a simple, non-replicated protocol that works well for data that is repeatedly accessed by a single processor, and then repeatedly accessed by other processors, such as data in producer-consumer applications. Neither the `conventional` nor the `migratory` protocols need twins or diffs.

The `write-shared` protocol supports multiple writers by creating page twins at the first write access by a processor, and creating and flushing diffs at the next release operation. The eager release consistent protocols used for comparison in this thesis were modeled after this protocol. However, our protocols use a static, distributed ownership scheme, while Munin's write-shared protocol uses dynamic page ownership.

Effectively, TreadMarks treats data as either `write-shared` (the default), synchronization (all synchronization goes through TreadMarks routines), or `read-only` (TreadMarks dynamically detects when pages are only accessed by a single processor and does not perform consistency action on these pages).

TreadMarks does not explicitly support `migratory` data, but the hybrid protocol effectively duplicates some of it's effects. Messages are only exchanged between the producer and the consumer, overlapped on the synchronization message if possible. However, all data in the TreadMarks system goes through the diffing mechanism, while in Munin `migratory` and `conventional` data does not.

TreadMarks has no need of Munin's timeout mechanism because the protocols are all invalidate-based.

As with many of the systems discussed in this chapter, direct comparison between Munin and TreadMarks performance is difficult because of differences in the underlying systems (Munin was implemented on top of 16 Sun 3/60s connected by a 10MBit ethernet and running the V [Che88] operating system).

Midway [ZSB94] and Concord are software DSMs based on *entry consistency* (EC). Entry consistency requires each shared data object to be attached to a synchronization object. On a lock acquisition, EC only propagates modified data associated with that lock, rather than attempting to use runtime information to dynamically predict which data will be useful, as in our hybrid (LH) protocol. The associations allow Midway to avoid moving more data than necessary, and to avoid access misses entirely.

The downside of this potential performance gain is that the memory model has substantially changed from the simple SC model, and the programmer is required to insert additional synchronization and data associations into shared memory programs. The explicit associations also force objects to be decomposed differently. On an SC or RC system, for instance, SOR's primary data structure is a single two-dimensional array. In order to run SOR on an EC system, border rows must be in different objects than rows in the interior of the set of rows assigned to a given process. Moreover, this example shows that the decomposition is even affected by the number or processes running the program.

Another novel idea in the Midway project was the use of software dirty bits to detect modifications to shared pages. A software dirty bit is associated with each cache line in the system. The compiler generates code to flip the associated dirty bit on each shared write. The bit-flipping adds approximately 10 cycles of overhead to each shared write. If this approach were used in our version of Water, the total overhead of detecting modifications might be less than 1% instead of 2.7%.

However, the use of software dirty bits has disadvantages as well. First, overhead is proportional to the number of shared writes, and not the total amount of modified data. Therefore, applications that modify pages repeatedly between synchronization transfers tend to perform worse with dirty bits than with diffing. Second, when a page is declared shared, but is effectively used as private storage, systems that use dirty bits still incur overhead on every write. With a lazy diffing mechanism, diffs are never created for these pages. Third, the current implementation by the Midway group relies heavily on several non-standard features of the DECStation's cache structure, and hence is not highly portable. Finally, the dirty bit approach requires significant compiler involvement, while the diff approach is language and compiler independent.

Concord [Lee94] extends entry consistency by allowing *handlers* to be associated with synchronization operations, as well as data. A handler is a procedure that is run at synchronization time, and specifies exactly what data should be updated. Since the handler is provided by the application, it allows update behaviors to be tailored very closely to specific applications. However, the use of handlers complicates the programming model even further.

Finally, the Carlos system [KFJ94] integrates message passing into an LRC system by providing a message interface and requiring messages to be annotated as to their significance to memory consistency. The message passing interface can be used to build efficient high level synchronization constructs such as queues or heaps. Early performance results of Carlos are encouraging, showing substantial gains over TreadMarks's performance for such lock-based programs as TSP, QS, and Water.

## 5.2 Hardware-Supported Shared Memory

DASH [LLG+90] is a cluster-based machine that uses a directory-based cache coherency protocol. DASH was the first system of any type to support release consistency. However, DASH uses a write invalidate protocol to maintain consistency. DASH does allow updates to be pipelined.

Sesame [WHL92] is a hardware network interface that implements *eager sharing* by selectively sending updates before they are requested. This approach is almost exactly opposite of LRC Their results indicate that eager sharing can perform an order of magnitude better than demand-driven systems under ideal conditions.

The SHRIMP [BLA+93] parallel machine consists of commodity workstations connected to a commodity backplanes via a custom network interface. The SHRIMP interface snoops the workstation buses and allows pages to be shared between pairs of machines via either automatic (any writes to shared data are immediately forwarded to the network) or deliberate (writes are only forwarded after an explicit command) updates. The sharing allows SHRIMP to emulate both low latency message passing and a form of shared memory. One of the primary innovations of SHRIMP is the separation of page *bindings* from data transfer. Binding together pages on two machines requires operating system intervention, but occurs only rarely. Once a binding has been established, the two machines can communicate entirely through user level messages. While SHRIMP is clumsy at best as a shared memory machine, the high performance communication mechanism would greatly facilitate the implementation of a high-performance software DSM.

## 5.3 Combined Hardware/Software Approaches

One of the most dominant trends in recently developed parallel architectures is the combination of software and hardware support for distributed shared memory.

Alewife [ALKK90, CA94] is a sequentially consistent system that uses directory-based cache coherence. Alewife uses a LimitLESS [CKA91] directory structure that supports a small number of directory pointers directly in hardware, and traps to a software handler when the pointers are exhausted. Simulations show that the LimitLESS directories achieve between 71% and 100% of the performance of full-map directory structures on a 256 node system. Another of Alewife's innovations is the use of very fast context switching of threads to hide access latency.

Architectures such as LimitLESS can benefit from the *Check-In/Check-Out* (CICO) [HLRW92] programming model. CICO allows programmers to pass performance directives directly to the memory system. In a CICO system, all accesses to shared memory are bracketed by `check_in` and `check_out` instructions. The first access to a piece of shared data is preceded by a `check_out` directive, while the last access to the data is followed by a `check_in` directive. The CICO directives let the system coordinate access to shared data with less communication than conventional systems, because at all times the system has exact knowledge of the set of processors accessing a given piece of shared data. The annotations can also be used to increase cache reuse and to reduce data sharing. CICO was developed with a shared memory system implementing a conventional memory model in mind. However, information from the directives could benefit an LRC system as well.

Typhoon [RLW94] supports shared memory over a message-passing network by using a communication co-processor to connect each primary processor to the network. All communication and protocol code is handled by the communication co-processor, freeing the main processor to continue executing application code.

Finally, Flash [Kea94] is an even more ambitious project that uses a custom designed node controller (MAGIC) to handle all communication both within and between nodes. MAGIC uses hardwired data paths to efficiently transfer most data without software intervention, but can be programmed to handle a variety of sophisticated protocols.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

The goal of this dissertation was to develop high-performance software DSM protocols that support a broad range of applications and achieve performance that approaches that of hardware. We have achieved this goal. This thesis presents two new DSM protocols: lazy invalidate (LI) and lazy hybrid (LH). Both of these protocols support an abstraction of shared memory that is indistinguishable from hardware supported shared memory for a broad class of applications. LH differs from LI in that it speculatively moves data with synchronization. We have shown that the performance of these software protocols is comparable to hardware supported performance in many cases, and substantially better than the best previous software protocols in most of the other cases. The following paragraphs summarize our main findings.

We first compared the performance of LI and LH with EI, a protocol representative of the previous state of the art. We found that seven of the eight applications in our suite perform better on the lazy protocols than on the eager, and four of those eight applications performed at least 18% better. Through a detailed breakdown of execution time we showed that the cost of executing our slightly more complicated protocol code was far outweighed by the cost of network communication in our system, and especially the software overhead involved in sending and receiving messages.

We then used a simulator to assess the effects of changing hardware and operating system overheads on the tradeoffs between the protocols. Both LH and EI improve relative to LI as software overhead is reduced. When per byte overhead is reduced to zero, EI outperforms both of the lazy protocols for four of the eight applications. Neither a reduction in the per message overhead nor an increase in network bandwidth allowed EI to outperform the lazy protocols on any application but FFT, but both changes decreased the gap in performance.

We then compared the performance of LH to LP, a variant of the hybrid protocol that uses user annotations instead of a run time heuristic to decide which data to speculatively move. LH outperformed LP for six of the eight applications. Our results show that the heuristic is usually much more effective than program annotations at identifying potentially useful data.

Finally, we compared the performance of our software protocols with that of a hardware shared memory system, and found that the difference in performance between the two systems correlates roughly with synchronization rate. The lazy system was able to approach, and in one case surpass, the hardware system's performance for coarse-grained applications. Fine-grained applications, however, performed much better on the hardware system. We extended these results with simulations that contrasted hardware and software implementations of shared memory with an intermediate approach that uses hardware and software techniques at different levels of the system. We found that the intermediate approach was able to scale with the hardware approach for most applications, degrading only for applications with fine-grained synchronization.

Overall, the results in this work show that lazy protocols generally require less communication than the previous state of the art, protocols implementing eager release consistency. LI and LH consistently outperformed EI across a wide variety of application types. The results show that software DSMs using protocols such as LI and LH have matured to the extent that they are now a truly viable alternative for high performance parallel computing.

## 6.2 Future Work

A key factor in determining both this work's long-term relevance and avenues for future research is the question of what types of parallel applications are expected to predominate in the future. Currently, static, data-parallel scientific code is far more common than any other type. Many of these applications exclusively use global synchronization such as barriers, and hence can not benefit from many of LRC's optimizations. Applications that use local synchronization are more amenable to optimization, but are less common. Of the five lock-based applications evaluated in this work, for example, four were written explicitly to test parallel programming systems. However, we expect applications with dynamic sharing patterns to become more common as parallel systems enter mainstream business.

We therefore expect the following two avenues for future research to be productive: (i) integrated parallel programming environments, and (ii) fault-tolerant DSMs.

### 6.2.1 Integrated Parallel Programming Environments

This research has shown that high performance can be achieved by protocols that are oblivious to application semantics. However, work with the hybrid protocol, as well as work by other researchers [KFJ94, ACDZ94], has shown that the protocols can greatly benefit from being more closely integrated with the application. We expect this direction of research to include two key components:

1. *De-constructing LRC* - By making more of the system visible to users at the application level, we expect to achieve a better match of system mechanisms to application semantics.

2. *Compiler Integration* - Compiler analysis can be useful in detecting specific access patterns and supplementing user annotations.

Other researchers [KFJ94] have shown that lazy release consistency can be *de-constructed* into building blocks that can be used to create custom synchronization mechanisms. We envision extending this de-construction to expose data movement, consistency management, as well as synchronization flow. These new mechanisms can provide users with a way to provide high level descriptions of applications to the system.

Compiler analysis can also be useful in describing data movement. Compiler analysis is often thought to be confined to the domain of applications where exact analysis is possible, such as Fortran applications. However, even inexact analysis can be very useful to DSM systems because it can be used to optimize data movement without affecting correctness. Input from the compiler can be limited to hints that improve efficiency when the hints are correct, and do not affect consistency even when they are not.

### 6.2.2 Fault Tolerance

As research prototypes become real tools that move into the realm of mainstream business use, fault tolerance becomes an important part of the services that the tools are expected to provide. Lazy release consistency provides new opportunities for closer integration of fault tolerant substrates into the DSM system because much of the information needed by fault tolerant systems is already available. Systems such as Manetho [EZ92a, EZ92b] rely on detailed message ordering information that can be re-constructed by systems that maintain *happened-before-1* information. Checkpointing overhead can be reduced by maintaining invariants assuring that portions of shared memory can be reconstructed on failure. Finally, the fact that all communication in a DSM system is produced by the system itself provides extensive opportunities to tune the fault tolerant mechanisms.

Combining both of these approaches into a single system will provide powerful opportunities to bring high performance and high usability to a wide spectrum of users.

# Bibliography


[ACDZ94] S.V. Adve, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Replacing locks by higher-level primitives. Technical Report TR94-237, Rice University, 1994.

[Adv93] S.V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin, Madison, December 1993.

[AH93] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[ALKK90] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

[BCCR91] R. Bryant, P. Carini, H.-Y. Chang, and B. Rosenburg. Supporting structured shared virtual memory under Mach. In *Proceedings of the 2nd Mach Usenix Symposium*, November 1991.

[BFS89] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[BGR+88] D. Black, D. Golub, R. Rashid, A. Tevanian, and M. Young. The Mach exception handling facility. *SigPlan Notices*, 24(1):45–56, May 1988.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–74, January 1987.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986. Special Issue of SIGPLAN Notices, Volume 21, Number 11, November, 1986.

[BKT92] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, June 1992.

[BL92] T. Ball and J. Larus. Optimally profiling and tracing programs. In *POPL92*, pages 59–70, January 1992.

[BLA+93] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. Technical Report CS-TR-487-93, Department of Computer Science, Princeton University, November 1993.

[BSF+91] W.J. Bolosky, M.L. Scott, R.P. Fitzgerald, R.J. Fowler, and A.L. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[BT88] 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.

[CA94] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

[CAL+89] 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.

[Car93] J.B. Carter. *Munin: Efficient Distributed Shared Memory Using Multi-Protocol Release Consistency.* PhD thesis, Rice University, October 1993.

[CBZ91] 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.

[CDJ+91] 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.

[CF89] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[Che88] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[CIS93] R. W. Cottingham Jr., R. M. Idury, and A. A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[CKA91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[CLBHL93] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

[Col91] W.W. Collier. *Reasoning about Parallel Architectures.* Prentice Hall, 1991.

[Cox92] A.L. Cox. *The Implementation and Evaluation of a Coherent Memory Abstraction for NUMA Multiprocessors.* PhD thesis, University of Rochester, Rochester, NY, May 1992.

[DCM+90] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wileknloh. The design and implementation of the Clouds distributed operating system. *Computing Systems Journal*, 3, Winter 1990.

[DKCZ93] 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.

[DSC+94] 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.

[EZ92a] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers Special Issue On Fault-Tolerant Computing*, 41(5):526–531, May 1992.

[EZ92b] E.N. Elnozahy and W. Zwaenepoel. Replicated distributed process in Manetho. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 18–27, July 1992.

[FP89] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[GLL+90] 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.

[HLRW92] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware support for scaleable multiprocessors. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[HWC+93] J. T. Hecht, Y. Wang, B. Connor, S. H. Blanton, and S. P. Daiger. Nonsyndromic cleft lip and palate: No evidence of linkage to hla or factor 13a. *American Journal of Human Genetics*, 52:1230–1233, 1993.

[JZ93] D.B. Johnson and W. Zwaenepoel. The Peregrine high-performance RPC system. *Software: Practice and Experience*, 23(2):201–221, February 1993.

[Kaa92] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, December 1992.

[KCZ92] 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.

[Kea94] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[KFJ94] Povl T. Koch, Robert J. Fowler, and Eric Jul. Message-driven relaxed consistency in a software distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 75–86, November 1994.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LE91] R. P. LaRowe and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[Lea87] E. L. Lusk and R. A. Overbeek et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc, 1987.

[Lee94] J. William Lee. Concord: Re-thinking the division of labor in a distributed shared memory system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 585–592, may 1994.

[LEK91] R. P. LaRowe, C. S. Ellis, and L. S. Kaplan. The robustness of numa memory management. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LLG+90] 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.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LLJO84] 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.

[LRC+92] A. Law, C. W. Richard III, R. W. Cottingham Jr., G. M. Lathrop, D. R. Cox, and R. M. Myers. Genetic linkage analysis of bipolar affective disorder in an old order amish pedigree. *Human Genetics*, 88:562–568, 1992.

[LT88] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel & Distributed Algorithms*. Elsevier Science Publishers, 1989.

[MF89] R.C. Minnich and D.J. Farber. The mether system: A distributed shared memory for SunOS 4.0. In *Proceedings of the 1989 Summer Usenix Conference*, pages 51–60, June 1989.

[MF90] R.C. Minnich and D.J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *Proceedings of the 10th Inter-*

*national Conference on Distributed Computing Systems*, pages 468–475, May 1990.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[SJG92] P. Stenström, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 1992.

[SLM90] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. Multi-model parallel programming in Psyche. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[SWG91] 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.

[WHL92] L.D. Wittie, G. Hermannsson, and A. Li. Eager sharing for efficient massive parallelism. In *1992 International Conference on Parallel Processing*, pages 251–255, St. Charles, IL, August 1992.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, October 1987.

[ZSB94] Mathew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.