Software Versus Hardware Shared-Memory Implementation: A Case Study

Alan L. Cox, Sandhya Dwarkadas, Pete Keleher,
Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel
Department of Computer Science
Rice University

Abstract

We compare the performance of software-supported shared memory on a general-purpose network to hardware-supported shared memory on a dedicated interconnect.

Up to eight processors, our results are based on the execution of a set of application programs on a SGI 4D/480 multiprocessor and on TreadMarks, a distributed shared memory system that runs on a Fore ATM LAN of DECstation-5000/240s. Since the DECstation and the 4D/480 use the same processor, primary cache, and compiler, the shared-memory implementation is the principal difference between the systems. Our results show that TreadMarks performs comparably to the 4D/480 for applications with moderate amounts of synchronization, but the difference in performance grows as the synchronization frequency increases. For applications that require a large amount of memory bandwidth, TreadMarks can perform better than the SGI 4D/480.

Beyond eight processors, our results are based on execution-driven simulation. Specifically, we compare a software implementation on a general-purpose network of uniprocessor nodes, a hardware implementation using a directory-based protocol on a dedicated interconnect, and a combined implementation using software to provide shared memory between multiprocessor nodes with hardware implementing shared memory within a node. For the modest size of the problems that we can simulate, the hardware implementation scales well and the software implementation scales poorly. The combined approach delivers performance close to that of the hardware implementation for applications with small to moderate synchronization rates and good locality. Reductions in communication overhead improve the performance of the software and the combined approach, but synchronization remains a bottleneck.

This research was supported in part by the National Science Foundation under Grants CCR-9116343, CCR-9211004, CDA-9222911, and CDA-9310073, and by the Texas Advanced Technology Program under Grants 003604014 and 003604012.

1 Introduction

Over the last decade, considerable effort has been spent on software implementations of shared memory on general-purpose networks, e.g., [2, 4, 18]. We are, however, unaware of any study comparing the performance of any of these systems to the performance of a hardware implementation of shared memory on a dedicated interconnect, e.g., [10, 17]. Several studies have compared software to hardware cache coherence mechanisms [20, 22], but these systems still rely on hardware initiated data movement and a dedicated interconnect. In this paper, we compare a shared-memory implementation that runs *entirely* in software on a general-purpose network of computers to a hardware implementation on a dedicated interconnect.

Up to eight processors, our results are based on an experimental comparison of a software and a hardware implementation. Specifically, we compare the TreadMarks software distributed shared memory system [15] running on a 100Mbit/second ATM network connecting 8 DECstation-5000/240s to an 8-processor Silicon Graphics 4D/480. These configurations have identical processors, clock speeds, primary caches, compilers, and parallel programming interfaces (the ANL PARMACS macros [19]). The similarity between the two platforms "from the neck up" avoids many distinctions that often blur comparative studies, and allows us to focus on the differences caused by the shared-memory implementation. TreadMarks supports *lazy release consistency* [14] and is implemented as a user-level library on top of Ultrix [15]. The SGI 4D/480 provides processor consistency, using a bus snooping protocol [21].

We use four applications in our comparison (ILINK, SOR, TSP, and Water). TSP uses only locks for synchronization, SOR and ILINK use only barriers, and Water uses both. For ILINK, SOR, and TSP, we present results for two different sets of input data. For Water, the results are largely independent of the input. Instead, we present the results for a modified version (M-Water) that reduces the amount of synchronization. With the exception of SOR, better speedups are obtained on the 4D/480. There is a strong correlation between the synchronization frequency of the application and the difference in speedup between the 4D/480 and TreadMarks. With higher synchronization frequencies, the large latencies of the software implementation become more of a limiting factor. SOR, however, gets better speedup on TreadMarks than on the 4D/480, because this application requires large memory bandwidths.

Beyond eight processors, our results are based on execution-driven simulations of systems with up to 64 processors. We compare three alternative designs: (i) an all-software (AS) approach connecting 64 uniprocessor machines with a general-purpose network, (ii) an all-hardware (AH) approach connecting 64 uniprocessor nodes with a crossbar network and using a directory-based hardware cache coherence protocol, and (iii) a hardware-software (HS) approach connecting 8 bus-based multiprocessor nodes with a general-purpose network and using the TreadMarks software DSM system. The HS approach is appealing from a cost standpoint because small bus-based shared memory workstations are likely to become cheaper than a set of uniprocessor workstations with an equal number of processors. The HS approach also avoids the complexity of directory-based cache controllers.

We use SOR, TSP, and M-Water in our comparison. Simulation times for available ILINK inputs were prohibitively high. For all three applications, the AS approach scales poorly compared to the other two. For SOR and TSP, performance of AH and HS is comparable. For M-Water, the AH approach performs significantly better because each processor accesses a majority of the shared data during each step of the computation and because of the frequency of synchronization. Anticipated improvements in network interface technology and attendant decreases in communication software overhead reduce the performance gap between the different implementations.

The rest of this paper is organized as follows. Section 2 details the comparison between the SGI 4D/480 and TreadMarks. Section 3 presents simulation results comparing the AS, AH, and HS architectures for a larger number of processors. Section 4 examines related work. Section 5 presents our conclusions.

2 SGI 4D/480 versus TreadMarks

2.1 TreadMarks

In this section we briefly describe the *release consistency* (RC) model [11] and the *lazy release consistency* (LRC) implementation [14] used by TreadMarks. Further details on TreadMarks may be found in Keleher et al. [15].

RC is a relaxed memory consistency model. In RC, *ordinary* shared memory accesses are distinguished from *synchronization* accesses, with the latter category subdivided into *acquire* and *release* accesses. Acquire and release accesses correspond roughly to the conventional synchronization operations on a lock, but other synchronization mechanisms can be built on this model as well. Essentially, RC allows the effects of ordinary shared memory accesses to be delayed until a subsequent release by the same processor is performed.

The LRC algorithm used by TreadMarks delays the propagation of modifications to a processor until that processor executes an acquire. To do so, LRC uses the *happened-before-1* partial order [1]. The *happened-before-1* partial order is the union of the total processor order of the memory accesses on each individual processor and the partial order of release-acquire pairs. Vector timestamps are used to represent the partial order [14]. When a processor executes an acquire, it sends its current vector timestamp in the acquire message. The last releaser then piggybacks on its response a set of *write notices*. These write notices describe the shared data modifications that precede the acquire according to the partial order. The acquiring processor then determines the pages for which the incoming write notices contain vector timestamps larger than the timestamp of its copy of that page in memory. For these pages, the shared data modifications described in the write notices must be reflected in the acquirer's copy. To accomplish this, TreadMarks invalidates the copies.

On an access fault, a page is validated by bringing in the necessary modifications to the local copy in the form of *diffs*. A diff is a run-length encoding of the changes made to a single virtual memory page. The faulting processor uses the vector timestamps associated with its copy of the page and the write notices it received for that page to identify the necessary diffs.

2.2 Experimental Platform

The system used to evaluate TreadMarks 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 25 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 workstations run the Ultrix version 4.3 operating system. TreadMarks is implemented as a user-level library linked in with the application program. No kernel modifications are necessary. TreadMarks uses conventional Unix socket, `mprotect`, and signal handling interfaces to implement communication and memory management. The minimum time for a remote lock acquisition is 0.78 milliseconds, the time for an 8-processor barrier is 2.20 milliseconds.

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, however, 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.

2.3 Application Suite

We used four programs for our comparison: ILINK, SOR, TSP, and Water.

ILINK [9] is a widely used genetic linkage analysis program that locates specific disease genes on chromosomes. We ran ILINK with two different inputs, CLP and BAD, both corresponding to real data sets used in disease gene location. The CLP and BAD inputs show the best and the worst speedups, respectively, among the inputs that are available to us.

Red-Black Successive Over-Relaxation (SOR) is a method for solving partial differential equations. The SOR program divides the matrix into roughly equal size bands of consecutive rows, assigning each band to a different processor. Communication occurs across the boundary between bands. We ran SOR for a 100 iterations on a 2000×1000 and a 1000×1000 matrix. We chose the 2000×1000 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 1000×1000 run is included to assess the effect of changing the communication to computation ratio.

TSP solves the traveling salesman problem using a branch-and-bound algorithm. The program has a shared, global queue of partial tours. Each process gets a partial tour from the queue, extends the tour, and returns the results back to the queue. We use 18- and 19-city problems as input. Although the program exhibits nondeterministic behavior, occasionally resulting in super-linear speedup, executions with the same input produce repeatable results.

Water, from the SPLASH suite [23], is a molecular dynamics simulation. The original Water program obtains a lock on the record representing a molecule each

Program	DEC	TreadMarks	SGI
ILINK-CLP	6385.0	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
Water-288-5	43.1	44.4	44.4
M-Water-288-5	43.1	43.7	44.1

Table 1: Single processor execution times

time it updates the contents of the record. We modified Water such that each processor instead uses a local variable to accumulate its updates to a molecule's record during an iteration. At the end of the iteration, it then acquires a lock on each molecule that it needs to update and applies the accumulated updates at once. The number of lock acquires and releases for each processor in M-Water is thus equal to the number of molecules that processor updates. In the original program, it is equal to the number of updates that processor performs, a much larger quantity. We present the results for Water and M-Water for a run with 288 molecules for 5 time steps. The results for Water were largely independent of the data set chosen.

2.4 Results

Figures 1 to 8 present the speedups achieved for ILINK, SOR, TSP, Water and M-Water, both on TreadMarks and the 4D/480. The TreadMarks speedups are relative to the single processor DECstation run times without TreadMarks. Table 1 presents the single processor execution times on both machines, including the DECstation with and without TreadMarks. As can be seen from this table, the presence of TreadMarks has almost no effect on single processor execution times. Finally, Table 2 details the off-node synchronization rates, the number of messages and the amount of data movement per second on TreadMarks for each of the applications on 8 processors. Sections 2.4.1 to 2.4.4 discuss the results for each application in detail.

2.4.1 ILINK

Figures 1 and 2 show ILINK's speedup for the CLP and BAD inputs. Among the inputs that are available to us, the CLP and BAD inputs show the best and the worst speedups, and the smallest and largest difference

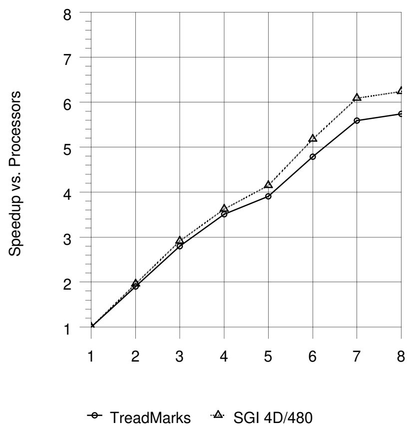

Figure 1: ILINK: CLP

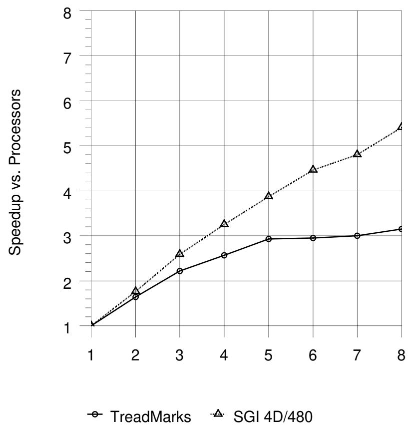

Figure 2: ILINK: BAD

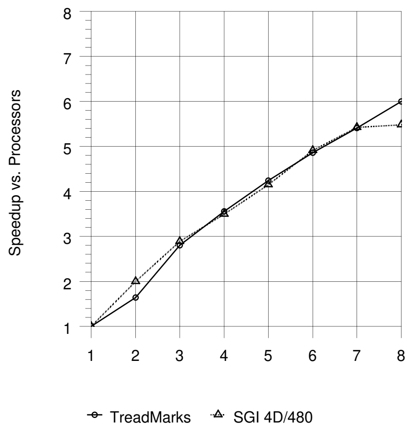

Figure 3: SOR: 2000 $\times$ 1000 matrix

	ILINK		SOR		TSP		Water	M-Water
	BAD	CLP	2000 × 1000	1000 × 1000	19-city	18-city	288/5	288/5
Barriers/second	10.14	0.36	2.89	4.41	—	—	0.45	3.51
Remote locks/second	—	—	—	—	14.6	32.3	1540.0	680.4
Messages/second	1800	449	100	154	407	536	6161	2739
Kbytes/second	538	161	17	30	126	223	717	936

Table 2: 8-processor TreadMarks execution statistics

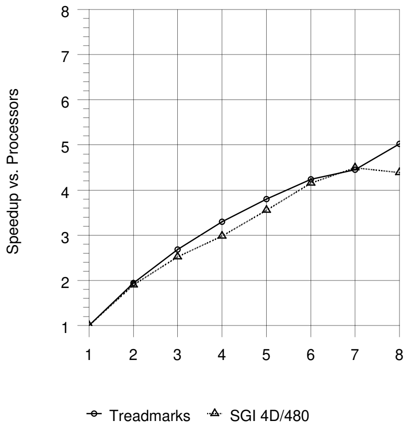

Figure 4: SOR: 1000 × 1000 matrix

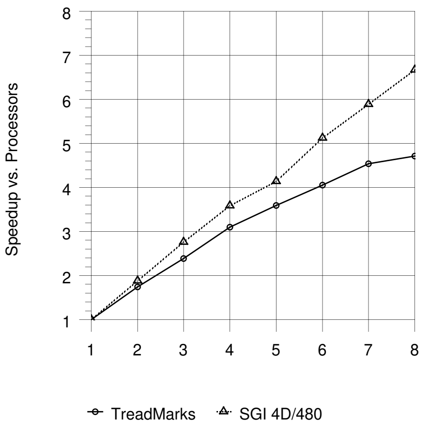

Figure 6: TSP: 18 cities

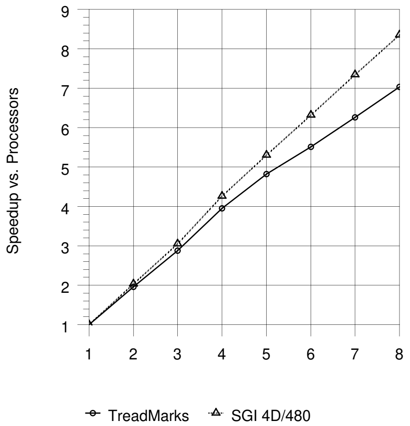

Figure 5: TSP: 19 cities

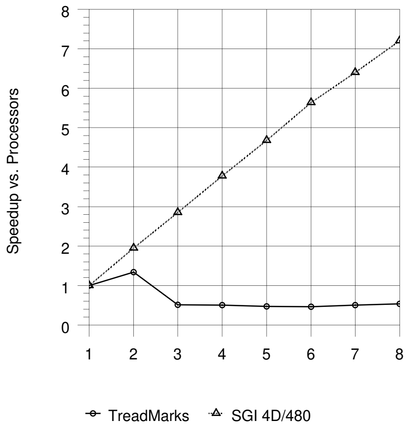

Figure 7: Water: 288 molecules and 5 steps

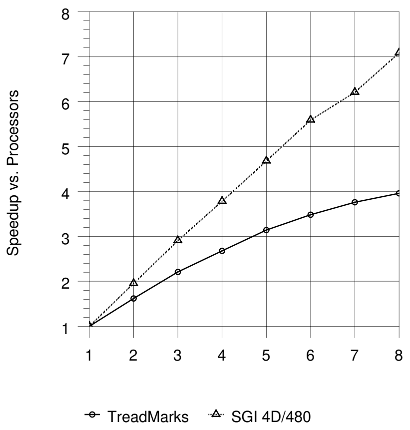

Figure 8: M-Water: 288 molecules and 5 steps

in speedup between the 4D/480 and TreadMarks (5.74 vs. 6.24 and 3.15 vs. 5.41).

ILINK achieves less than linear speedup on both the 4D/480 and TreadMarks because of a load balancing problem inherent to the nature of the algorithm [9]. It is not possible to predict in advance whether the set of iterations distributed to the processors will result in the same amount of work on each processor, without significant computation and communication.

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.

2.4.2 SOR

Figures 3 and 4 show SOR's speedup for 100 iterations of 2000×1000 and 1000×1000 problems. We excluded the first iteration of SOR from the data and message rates in order to avoid having the initial data distribution skew our results. 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 Section 2.1) 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 for 8 processors 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 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. On the ATM network, this communication can occur in parallel. On the 4D/480, it causes contention for the cache tags and the bus.

2.4.3 TSP

Figures 5 and 6 show TSP's speedup for solving a 19-city and an 18-city problem. 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 (see Table 2).

The performance on TreadMarks suffers from the fact that TSP is not a *properly labeled* [11] program. 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 since 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 nondeterministic effect of picking up redundant work due to the slight delay in propagating the bound.

2.4.4 Water

Figure 7 shows Water's speedup executing 5 steps on 288 molecules. TreadMarks gets no speedup, except on 2 processors, because the high rate of synchronization (1,540 remote lock acquires/second) causes many messages (6,161 messages/second).

Figure 8 shows M-Water's speedup executing 5 steps on 288 molecules. On the 4D/480, M-Water's speedup is virtually identical to Water. On TreadMarks, however, there is a marked performance improvement. We obtain a speedup of 3.96 using 8 processors. Compared to Water, the number of messages/second drops from 6,161 to 2,739.

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 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 M-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.

2.5 Summary

The relative magnitude of the differences in speedup between TreadMarks and the 4D/480 for ILINK, TSP, Water and M-Water roughly correlate to the differences in the synchronization rates. For TSP, Water and M-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 Water (with 1540 remote lock accesses per second), 3.2 for M-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, the difference in speedup can be explained by the barrier synchronization frequency, a difference of 2.2 for the BAD data set with 10.14 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 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 results for M-Water demonstrate the possible performance improvement.

3 Comparison of Larger Systems

In this section, we extend our results to larger numbers of processors. The software approach scales, at least conceptually, to a larger number of processors without modification. The hardware approach, however, becomes more complex once the number of processors exceeds what can reasonably be supported by a single bus. The processor interconnect instead becomes a mesh or a crossbar with one or more processors at the nodes, and the cache controllers implement a directory-based cache coherence protocol. We also model a third architecture that consists of a number of bus-based multiprocessors connected by a general-purpose network. Each node has sufficient bus bandwidth to avoid contention. Conventional bus snooping hardware enforces coherence between the processors within a node. Coherence between different nodes is 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 appears promising both in terms of cost and 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.

3.1 Simulation Models

We simulated the three architectures using an execution-driven simulator [7]. 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 18 processor cycles, 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 has been modified. These cycle counts are similar to those for the Stanford DASH [17] and FLASH [16] 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 latency of 1 microsecond. Memory consistency between the nodes is maintained using the TreadMarks LRC invalidate protocol (see Section 2.1). 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 described in Section 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. Similarly, modifications to the same page made by processors on the same node are merged into a single diff. 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.

3.2 Results

We simulated SOR, TSP, and M-Water. Excessively long simulation times prevented us from including simulation results for ILINK. Figures 9 to 11 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 12 and 13 present the message and data movement totals for AS and AH. The totals are presented relative to the AS numbers. The message totals are broken down into miss and synchronization messages, and the data totals are broken down into miss, consistency, and header data (see Section 2.1). Sections 3.2.1 to 3.2.3 discuss the observed performance of the individual applications. Section 3.2.4 discusses the effect of reducing the software overhead for the AS and HS architectures.

3.2.1 SOR

Figure 9 presents speedups for the SOR program for a 2000×1000 matrix. Since we only simulate a small number of iterations, we begin collecting statistics from the second iteration in order to prevent cold start misses from dominating our results. Linear speedup is

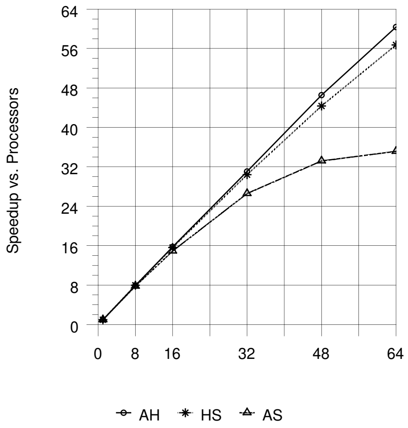

Figure 9: SOR: 2000 × 1000

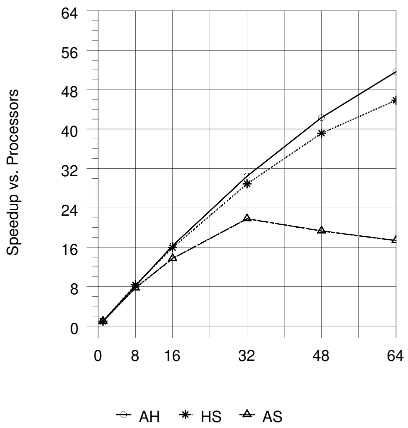

Figure 10: TSP: 19 cities

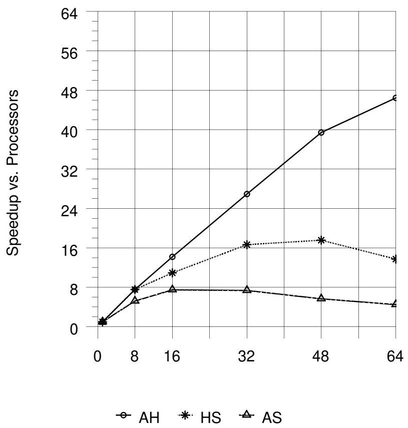

Figure 11: M-Water: 288 molecules and 2 steps

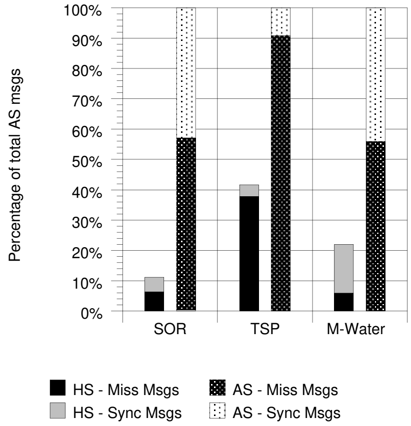

Figure 12: Total messages (64 processors)

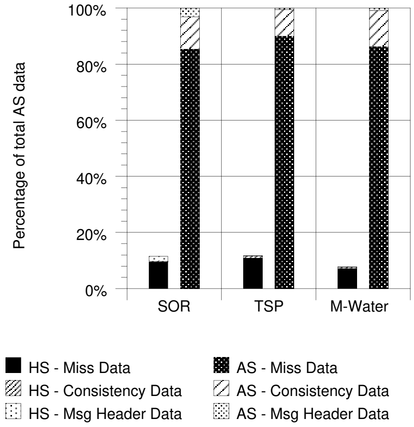

Figure 13: Total data (64 processors)

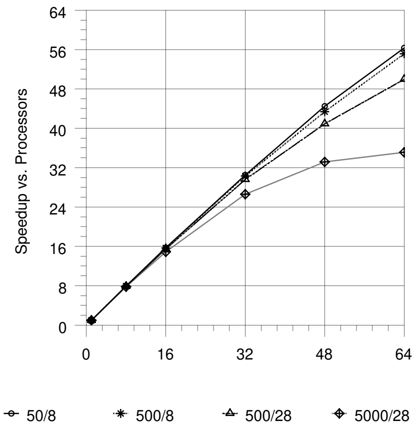

Figure 14: Varying software overheads for AS, SOR: 2000 × 1000

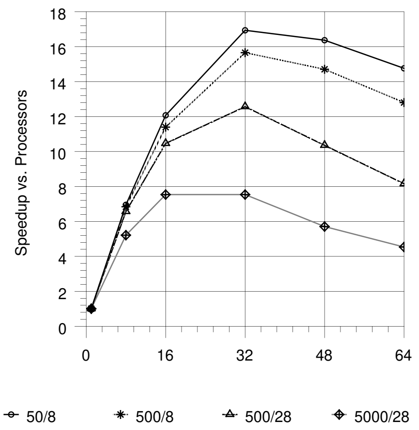

Figure 15: Varying software overheads for AS, M-Water: 288 molecules and 2 steps

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 and thus takes 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 12).

3.2.2 TSP

Figure 10 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 12 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 13 shows that the amount of data movement by HS 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 (see Section 3.1).

3.2.3 M-Water

Figure 11 presents speedups for M-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 7.49 at 16 processors, and HS reaches its peak speedup of 16.63 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. The HS approach gets better performance primarily because of the reduction in the number of messages. Also, less data is sent because of the coalescing of diffs and the reduction in the amount of consistency data. Although HS gets a 5-fold decrease in the *overall* number of messages and a 13-fold decrease 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 remain high (see Figure 12).

3.2.4 Reduced Software Overhead

By optimizing the software structure, as in Peregrine [13], or a user-level hardware interface, as in SHRIMP [3], lower software overheads can be achieved. 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 (as in Peregrine), and 50 (as in SHRIMP), and the per word cost from 28 processor cycles to 8, one `bcopy` to the interface.

Figures 14 and 15 present the speedups for SOR and M-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. The resulting speedup approaches that of the other two architectures. For M-Water, both the fixed and per word cost have equal effects on performance.

Figure 16 presents the speedups for M-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.

3.3 Summary

We conclude that the AS approach does not scale well for the applications and problem sizes that we simulated, unless the software overheads are significantly reduced. For SOR and TSP, the HS performance is almost identical to the AH approach. For

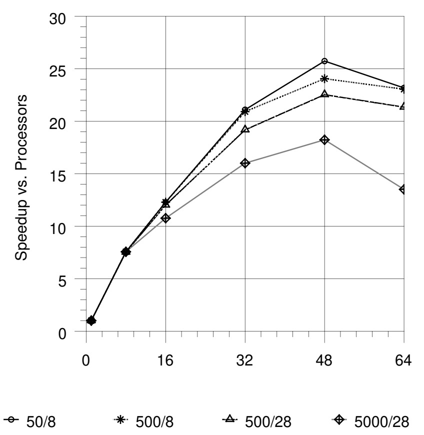

Figure 16: Varying software overheads for HS, M-Water: 288 molecules and 2 steps

Water, the frequent synchronization results in inferior performance for HS compared to AH. Our results are, of course, limited by the applications we simulated. Due to simulation time constraints, the problem sizes are small. The effect of larger applications remains to be investigated.

4 Related Work

TreadMarks implements shared memory *entirely* in software. Both data movement and memory coherence are performed by software using the message passing and virtual memory management hardware. Previous evaluations of such systems, for example, Carter *et al.* [4] have compared their performance to hand-coded message passing.

Other related studies have examined software versus hardware cache coherence. In these studies, the hardware is responsible for performing the data movement. Upon access, the hardware automatically loads invalid cache lines from memory. To maintain coherency, these schemes require the placement of cache flush/invalidation instructions by the compiler or the programmer at the end of critical sections. Cytron *et al.* [8] and Cheong and Veidenbaum [6] describe algorithms for compiler-based software cache coherence. Owicki and Agarwal compare analytically the performance of such a scheme to snoopy cache coherence hardware [20]. Petersen, on the other hand, describes a software cache coherence scheme using the virtual memory management hardware [22]. This scheme is transparent to the programmer. It does not require the programmer or compiler to insert cache flush instructions. Using trace-driven simulation, she compared the performance of her software scheme on a shared-bus to snoopy cache hardware.

A few implementations using both hardware and software have been proposed. Both Chaiken *et al.* [5] and Hill *et al.* [12] describe shared memory implementations that handle the most common cache coherence operations in hardware and the most unusual operations in software, thereby reducing the complexity of the hardware without significantly impacting the performance.

5 Conclusions

In this paper we have assessed the performance tradeoffs between hardware and software implementations of shared memory.

For small numbers of processors we have compared a bus-based shared memory multiprocessor, the SGI 4D/480, to a network of workstations running a software DSM system, specifically the TreadMarks DSM system running on an ATM network of DECStation-5000/240s. An important aspect of this comparison is the similarity between the two platforms in all aspects (processor, cache, compiler, parallel programming interface) except the shared memory implementation.

For the applications with moderate synchronization and communication demands, the two configurations perform comparably. When these demands increase, the communication latency and the software overhead of TreadMarks causes it to fall off in performance. For applications with high memory bandwidth requirements, the network of workstations can perform better because it provides the processor with a private path to memory.

We use simulation to extend our results to larger numbers of processors. For the sizes of the applications we considered, a straightforward extension of the software DSM system scales poorly unless software overheads reflect recent advances in communication software and hardware. We investigated an intermediate approach, using a general purpose network and software DSM to interconnect hardware bus-based multiprocessor nodes. Such a configuration can be constructed with commodity parts, resulting in cost and complexity gains over a hardware approach that uses a dedicated interconnect and a directory-based cache controller. For applications with good locality and moderate synchronization rates, the combined hardware-software approach results in performance comparable to that obtained using a pure hardware approach.

Acknowledgements

We thank Michael Scott at the University of Rochester and Michael Zeitlin at Texaco for providing us with access to their Silicon Graphics multiprocessors. We would also like to thank Benhaam Aazhang, John Bennett, Bob Bixby, Keith Cooper and John Mellor-Crummey for providing us with the extra computing power that we needed to complete the simulations.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] H. Cheong and A.V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 138–145, June 1988.

[7] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, January 1991.

[8] R. Cytron, S. Karlovsky, and K.P. McAuliffe. Automatic management of programmable caches. In *1988 International Conference on Parallel Processing*, pages 229–238, August 1988.

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

[10] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Technical report, Silicon Graphics Computer Systems, 1993.

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

[12] 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.

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

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

[15] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[16] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. To appear in Proceedings of the 21st Annual International Conference on Computer Architecture, April 1994.

[17] 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.

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

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

[20] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, May 1989.

[21] M. Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, May 1984.

[22] K. Petersen. *Operating System Support for Modern Memory Hierarchies*. PhD thesis, Princeton University, May 1993.

[23] 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.