Hiding Communication Latency and Coherence Overhead in Software DSMs *

R. Bianchini, L. I. Kontothanassis†, R. Pinto,
M. De Maria, M. Abud, and C. L. Amorim

COPPE Systems Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil 21945-970

†Department of Computer Science
University of Rochester
Rochester, New York 14627

Abstract

In this paper we propose the use of a PCI-based programmable *protocol controller* for hiding communication and coherence overheads in software DSMs. Our protocol controller provides three different types of overhead tolerance: a) moving basic communication and coherence tasks away from computation processors; b) prefetching of diffs; and c) generating and applying diffs with hardware assistance. We evaluate the isolated and combined impact of these features on the performance of TreadMarks. We also compare performance against two versions of the Shrimp-based AURC protocol. Using detailed execution-driven simulations of a 16-node network of workstations, we show that the greatest performance benefits provided by our protocol controller come from our hardware-supported diffs. Reducing the burden of communication and coherence transactions on the computation processor is also beneficial but to a smaller extent. Prefetching is not always profitable. Our results show that our protocol controller can improve running time performance by up to 50% for TreadMarks, which means that it can double the TreadMarks speedups. The overlapping implementation of TreadMarks performs as well or better than AURC for 5 of our 6 applications. We conclude that the simple hardware support we propose allows for the implementation of high-performance software DSMs at low cost. Based on this conclusion, we are building the NCP_2 parallel system at COPPE/UFRJ.

*C. L. Amorim is currently a visiting professor at the University of Rochester. This work was supported in part by FINEP/Brazil grant no. 56/94/0399/00, CNPq/Brazil, NSF Institutional Infrastructure grant no. CDA-8822724, and ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology – High Performance Computing, Software Science and Technology Program, ARPA Order no. 8930).

1 Introduction

Software distributed shared-memory systems (DSMs) provide programmers with the illusion of shared memory on top of message-passing hardware. These systems provide a low-cost alternative for shared-memory computing, since they can be built with standard workstations and operating systems. However, several applications running on software DSMs suffer high communication and coherence-induced overheads that limit performance.

In this paper we propose the use of a PCI-based programmable *protocol controller* for hiding or tolerating these communication and coherence overheads. In our system each protocol controller is associated with a single computation processor and provides three different types of overhead tolerance: a) moving basic communication and coherence tasks, such as diff generation and application, away from computation processors; b) prefetching of encodings of modifications made to pages (also called "diffs"); and c) generating and applying diffs with hardware assistance. In this paper we evaluate the isolated and combined impact of these features on the performance of TreadMarks [14].

The performance of the overlapping versions of TreadMarks is also compared to two versions of AURC, a software DSM based on automatic updates and optimized pair-wise sharing, as afforded by the Shrimp network interface [4]. Automatic updates are another form of overlapping overheads and useful computation, and can be used to obviate the need for diffs. In the presence of pair-wise sharing, automatic updates can also reduce the number of times pages must be fetched. Both automatic-update-based DSMs we implement include these two optimizations, while one of them also implements prefetching.

Using detailed execution-driven simulations of a 16-node network of workstations, we show that the greatest performance benefits provided by our protocol controller come from generating and applying diffs with hardware assistance. Reducing the burden of communication and coherence transactions on the computation processor is also profitable but improvements are not as significant and consistent as with our hardware-supported diffs. Prefetching is not always beneficial, as it may increase synchronization latency and interprocessor interference more than reduce communication over-

head. Our results also show that our protocol controller can improve running time performance by up to about 50% for TreadMarks, which means that it can double the TreadMarks speedups. The overlapping implementation of TreadMarks performs as well or better than AURC for 5 of our 6 applications. In addition, we find that network and memory parameters have an important role in this comparison; a network with low bandwidth or a high messaging overhead can hurt AURC severely, while memories with low bandwidth and/or high latency can degrade the performance of our overlapping TreadMarks implementations significantly.

In contrast with previous approaches that provide a shared-memory abstraction with custom, highly-integrated support for cache coherence [19, 1, 18, 21], our design builds mainly upon off-the-shelf workstation parts and low-cost commercial networks. In contrast with systems that maintain cache coherence entirely in software [23] or with hardware support [17, 22], our system does not stress the communication medium and therefore is appropriate for low-cost, PCI-based networks exhibiting relatively high latency and low bandwidth. Thus, we conclude that our design provides a low-cost, quick-design-time alternative for shared-memory computing.

Based on this conclusion, we are building the NCP_2 parallel system at COPPE/UFRJ. In fact, the architecture we study in this paper resembles the first version of our system, NCP_{2s}, in which the protocol controller is not completely decoupled from the rest of the workstation hardware. In our second prototype (which is currently under design), the protocol controller card will provide the same functionality as in the NCP_{2s} but will only connect to the PCI bus, thus becoming virtually platform-independent.

In summary, this paper makes the following contributions.

- It proposes the design and implementation of simple hardware to hide communication and coherence overheads in software DSMs.
- It proposes extensions to TreadMarks that take advantage of the new hardware and evaluates the extensions' impact on performance.
- It presents an evaluation of standard TreadMarks in the presence of a runtime-based prefetching strategy. In addition, it presents the first evaluation of the combination of automatic updates, optimized pairwise sharing, and page prefetching.
- It evaluates the sensitivity of both the overlapping TreadMarks and AURC to several architectural parameters, including network and memory bandwidths.

The remainder of this paper is organized as follows. Section 2 overviews the main characteristics of software DSMs and their sources of overhead. Section 3 describes the hardware of our controller and how software DSMs can take advantage of it. Section 4 presents our methodology and application workload. Experimental results are presented in section 5. Section 6 summarizes the related work. Finally, section 7 summarizes our findings and concludes the paper.

2 Overheads in SW DSMs

Several software DSMs use virtual memory protection bits to enforce coherence at the page level. In order to minimize the impact of false sharing, these DSMs seek to enforce memory consistency only at synchronization points, and allow multiple processors to write the same page concurrently [6].

TreadMarks is an example of a system that enforces consistency lazily. In TreadMarks, page invalidation happens at lock acquire points, while the modifications (diffs) to an invalidated page are collected from previous writers at the time of the first access (fault) to the page. The modifications that the faulting processor must collect are determined by dividing the execution in *intervals* associated with synchronization operations and computing a *vector timestamp* for each of the intervals. A synchronization operation initiates a new interval. The vector timestamp describes a partial order between the intervals of different processors. Before the acquiring processor can continue execution, the diffs of intervals with smaller vector timestamps than the acquiring processor's current vector timestamp must be collected. The previous lock holder is responsible for comparing the acquiring processor's current vector timestamp with its own vector timestamp and sending back write notices, which indicate that a page has been modified in a particular interval. When a page fault occurs, the faulting processor consults its list of write notices to find out the diffs it needs to bring the page up-to-date. It then requests the corresponding diffs and waits for them to be (generated and) sent back. After receiving all the diffs requested, the faulting processor can then apply them in turn to its outdated copy of the page. A more detailed description of TreadMarks can be found in [15].

The main overheads in software DSMs are related to communication latencies and coherence actions. Communication latencies cause processor stalls that degrade system performance. Coherence actions (diff generation and application, twin generation, and directory manipulation) can also negatively affect overall performance, since they accomplish no useful work and are in the critical path of the computation. The impact of communication and coherence overheads is magnified by the fact that remote processors are involved in all of the corresponding transactions.

To demonstrate the extent of the overhead problem, consider figures 1 and 2[1]. Figure 1 presents the speedups achieved by our applications under the non-overlapping version of TreadMarks. The figure shows that our applications exhibit a broad range of speedups; from the unacceptable performance of Ocean to the reasonably good speedups of TSP.

Figure 2 presents a detailed view of the running time performance of our applications under the non-overlapping TreadMarks on 16 processors. The bars in the figure show normalized execution time broken down into busy time, data fetch latency, synchronization time, IPC overhead, and other overheads. The latter category is comprised by TLB miss latency, write buffer stall time, interrupt time, and the most significant of these overheads, cache miss latency. The busy time represents the amount of useful work performed by the

[1]The details of the simulation and application characteristics that led to these figures will be presented in section 4.

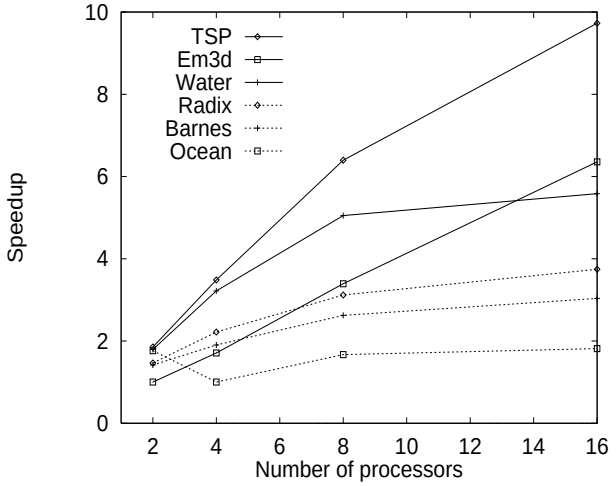

Figure 1: Application Speedups under TreadMarks DSM.

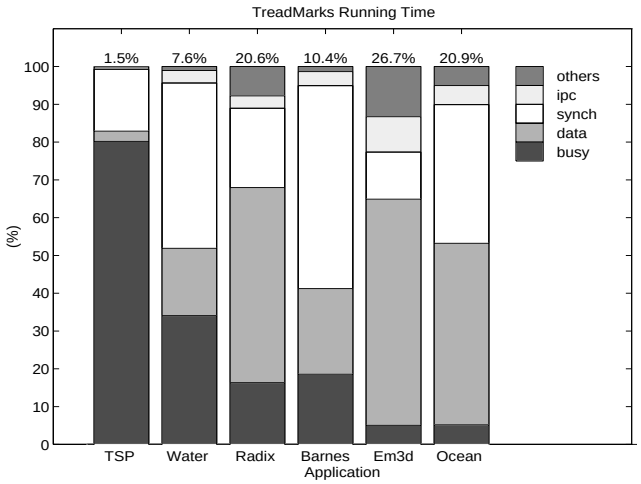

Figure 2: Application Performance under TreadMarks DSM on 16 processors.

computation processor. Data fetch latency is a combination of coherence processing time and network latencies involved in fetching pages and diffs as a result of page faults. Synchronization time represents the delays involved in waiting at barriers and lock acquires/releases, including the overhead of interval and write notice processing. IPC overhead accounts for the time the computation processor spends servicing requests coming from remote processors. The number on top of each bar represents the amount of time each processor spends on diff-related operations (twinning and diff generation and application) as a percentage of its execution time.

This figure shows that TreadMarks suffers severe data fetch and synchronization overheads. IPC overheads are not as significant since they are often hidden by data fetch and synchronization latencies. However, IPC overheads gain importance when prefetching is used. The amount of time each processor spends on diff-related operations is also significant; these operations can take more than 20% of the total execution time as in the case of Em3d, Ocean, and Radix.

The hardware support we propose, in the form of protocol controllers, attempts to alleviate these overheads by moving them out of the computation processor, hiding them behind useful computation when possible, and providing hardware support for generating and applying diffs. Obviously, techniques for tackling overheads are likely to achieve greater performance improvements for overhead-dominated applications. In the following sections, we will show that the techniques we study can double some of the speedups in figure 1.

3 Hiding Overheads in SW DSMs

In the previous section we saw that software DSMs suffer from various forms of overhead that limit performance. In this section we describe the hardware support we propose for reducing overheads or hiding them behind useful computation, and explain how software DSMs can efficiently use it.

3.1 Hardware for Hiding Overheads

As shown in figure 3, the hardware support we propose is a protocol controller associated with each of the computation processors in a network of workstations. In our system both the protocol controller and the network interface are plugged onto the PCI bus of a (commercial) workstation board. As shown in figure 4, our protocol controller includes a microprocessor (or simply an integer RISC core), 4 MBytes of DRAM, and two pieces of custom hardware: the logic for snooping the memory bus and a DMA engine.

Communication between a computation processor and its controller occurs via the controller's memory, interrupts, and snooping of write accesses. The controller's memory stores the protocol software, a command queue, and a table for translating virtual to physical addresses and recording status information. Whenever the computation processor needs a protocol-related service, it issues an explicit request to its protocol controller and continues its normal execution, unless the completion of the operation is required immediately [2]. This local command (and any commands coming from remote controllers) is placed in the controller's queue.

A computation processor may also require service by a remote node. These inter-node transactions are split into three different commands: a request issued by the computation processor to its local controller, an action by the remote controller/computation processor, and a reply issued by the remote controller. These commands are interleaved with other transactions in the command queues to improve controller throughput and occupancy. The protocol software keeps track of pending remote requests. At the time of the completion of one of these requests, the controller software must set a status bit to indicate the occurrence of this event. The pending requests and the bit flags can be used by the computation processor on page faults, so that it does not fetch data for which a prefetch is in progress or for which a

[2]Requests may be given high or low priority, so that we can prevent prefetches from delaying requests for which a computation processor is stalled waiting.

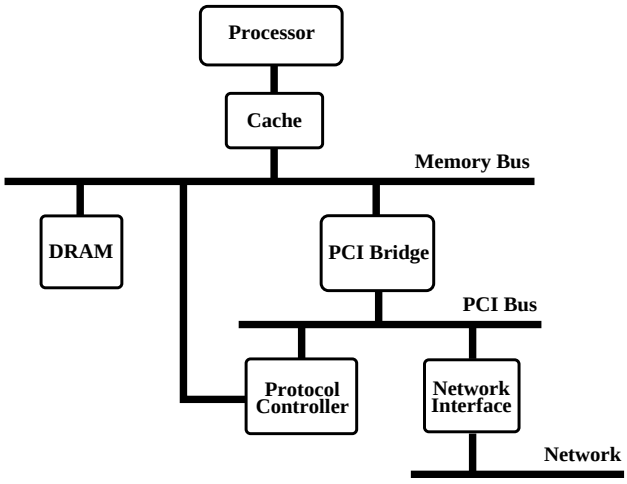

Figure 3: Node Architecture.

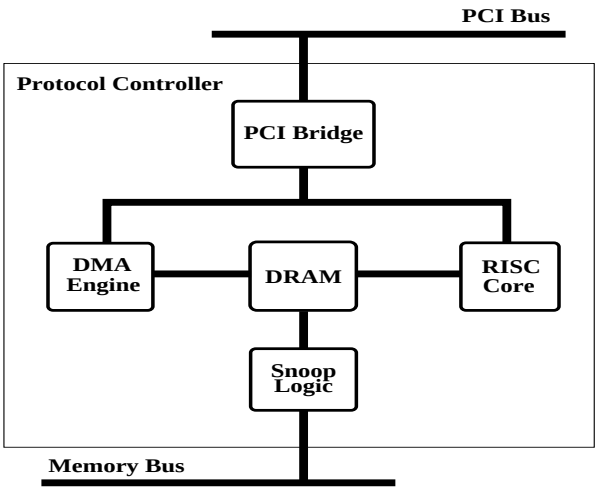

Figure 4: Zoom of Controller Architecture.

prefetch has completed, for instance. The virtual-to-physical address translation table is used by the controller software to decide where in main memory to store pages and diffs.

The protocol controller can also communicate with the computation processor by interrupting it when necessary. Computation processor interrupts should be avoided as much as possible for efficiency, but may be used to prevent complicated operations from crossing the PCI bus in order to reach main memory. In fact, the ratio between computation and controller processor speeds is also an issue here; a much faster computation processor provides another justification for running complicated protocol code on it.

Both the processor and its protocol controller snoop on the memory bus. The processor must snoop so that it can invalidate data written by the protocol controller directly to local memory, in the event of the application of a remote diff to a local page, for instance. The protocol controller must snoop so that it can compute diffs on-the-fly. This is accomplished by forcing the cache to write shared data through to the bus and keeping a record (in the controller's memory) of all the modified words in a page. The record is kept in the form of a bit vector, where each bit represents a word of data. Whenever the protocol controller sees that the computation processor has written a word, it simply sets the corresponding bit to record the event. Later, when the controller is asked for the page's diff, our custom DMA engine checks all the bits that are set, reads the corresponding words of memory, and returns the words and the bit vector as the page's diff. The diff can then be saved in main memory for later use. Generating the diff resets all the bits in the vector. Applying a diff also involves the DMA engine, which is used to assign the words stored in a diff to the destination page according to the diff's bit vector. Thus, our DMA engine simply performs scatter/gather operations directed by bit vectors.

The DMA engine takes a variable amount of time to determine all the words written in a page. As examples, it takes this hardware about 200 protocol controller cycles to scan the bit vector for a 4-Kbyte page if none of the words has been written, while it takes the hardware around 2100 cycles to scan the vector if all of its words have been written. In a standard software DSM these operations take about 7K cycles just for processor instructions. This simple comparison shows that our hardware support reduces the overall time required to generate diffs, besides obviating the need for twins and allowing the computation processor to perform useful work while coherence actions are taking place.

Overall, the hardware we propose is fairly simple as shown in figure 4. The only custom pieces of hardware involved in our architecture are the logic for snooping the memory bus and setting the bit vectors, and our DMA engine.

3.2 Using Protocol Controllers

The protocol controller we implemented for our experiments provides its computation processor with the basic mechanisms used by software DSMs. More specifically, the functionality offered by the controller we evaluate is: a) remote page request and reply; b) remote diff request and reply; c) local (on-the-fly) diff creation and application; d) message send and receive. The remaining tasks associated with protocol processing are usually more complicated, and thus run on the computation processor.

Our protocol controller provides software DSMs with numerous possibilities for hiding overheads. In our experiments, we concentrated on three different forms of overhead tolerance for our overlapping TreadMarks implementation:

- The basic communication and coherence actions, such as diff generation and application, are performed by the protocol controllers without burdening their associated computation processors. The computation processor must be interrupted under several circumstances, but the more expensive protocol actions are always executed by the controller. For instance, remote diff requests must interrupt the processor so that it can perform interval processing, but the diffs themselves are generated by the controller. This strategy allows for overlapping time-consuming protocol actions on the controller with useful work on the computation processor.
- Prefetching of diffs is accomplished by anticipating future accesses to shared data based on past history and

requesting them at the time of lock acquisitions. The write notices are used to determine the processors that must provide the diffs. This prefetching heuristic assumes that a computation processor will likely need pages it used to cache and reference, but were invalidated by another processor. Prefetching overlaps communication and coherence latencies associated with access faults with useful computation.

- Modified words are dynamically marked as pages are modified, while our DMA engine is used to generate and apply diffs. This scheme is supported directly by the hardware of our protocol controllers and not only obviates the need for using twins in our overlapping DSMs, but also significantly decreases the diff generation and application times and the local bus utilization.

3.3 Using Automatic Updates

For comparison purposes, we also study the performance of AURC, a software DSM based on Shrimp-style automatic updates and optimized pairwise sharing. In the Shrimp multicomputer [4], two user processes can communicate efficiently via messages by creating a mapping between the sender's virtual memory and the destination's virtual memory across the network. Write accesses to the sender's virtual memory are snooped by the network interface, which automatically propagates them to the destination's memory. The network interface combines consecutive updates in a way that resembles a write cache [9] to reduce network traffic. Since the network interface transfers updates while the source and destination processors proceed with their normal execution, automatic updates can be considered another form of overlapping overheads and useful computation.

Shrimp allows for optimized pairwise sharing of a page, since two processors can create a bi-directional mapping of the page so that all changes made by one processor can be seen by the other. Note that under this scheme, page faults and page fetches need not occur. To avoid initialization effects, the third processor to access the page simply replaces the first processor in the pairwise sharing scheme. If after that point more processors join the sharing set, the system reverts to write-through to the home node, by all processors.

In AURC pages shared by two processors are mapped bi-directionally as described above. Pages used by more than two processors are assigned home nodes, which store both data and directory information. Write accesses directed to shared pages are automatically forwarded to their respective home nodes, where modifications are merged. At a lock release operation, a processor has to send *flush timestamps* with an interval number across all links that were active during the interval and save this interval number in the *lock timestamp* of the pages it modified. Lock timestamps are passed along with lock ownership. At a lock acquire, the acquiring processor must invalidate the pages that the previous lock owner determined are out-of-date and update their corresponding lock timestamps. A page fault always requires the offending processor to wait until it has received all updates destined to it but still in progress and to fetch the page from the home node. Whether a processor has to wait for these updates depends on the difference between flush and lock timestamps. Further details about AURC can be found in [12].

AURC can potentially be improved by prefetching. Our prefetching implementation of AURC uses the same heuristic to initiate prefetches as the overlapping TreadMarks. Thus, at lock acquire points, invalidated pages that were cached and referenced are fetched prior to being referenced again.

In section 5, we isolate the performance improvement provided by each of the forms of overlapping we used in TreadMarks. In addition, we compare the performance of these protocols against AURC and AURC with prefetching.

4 Methodology and Workload

4.1 Simulation Infrastructure

Our simulator consists of two parts: a front end, Mint [24], that simulates the execution of the computation processors, and a back end that simulates the protocol controller and the memory system (finite-sized write buffers and caches, TLB behavior, full protocol emulation, network transfer costs including contention effects, and memory access costs including contention effects) in great detail. The front end calls the back end on every data reference (instruction fetches are assumed to always be cache hits). The back end decides which computation processors block waiting for memory (or other events) and which continue execution. Since this decision is made on-line, the back end affects the timing of the front end, so that the interleaving of instructions across processors depends on the behavior of the memory system and control flow within a processor can change as a result of the timing of memory references.

We simulate a network of workstations with 16 nodes in detail. Each node consists of a computation processor, a write buffer, a first-level direct-mapped data cache (all instructions are assumed to take 1 cycle), local memory, a mesh network router (using wormhole routing), and the custom protocol controller. The network interface and the protocol controller reside on the PCI bus, which is also fully modeled. The protocol controller is assumed to have an option for creating diffs statically (and therefore managing twins), for comparison purposes. Table 1 summarizes the default parameters used in our simulations; variations of these parameters are studied in section 5.3. All times are given in 10-ns processor cycles. The processor and the DMA engine in the protocol controller are assumed to run at the same speed as the computation processor.

4.2 Workload

Our application suite includes applications exhibiting widely different speedup levels as shown in section 2. We report results for six parallel programs: TSP, Barnes, Radix, Water, Ocean, and Em3d. TSP is from Rice University and comes with the TreadMarks distribution. The next four applications are from the Splash-2 suite [25]. These applications were run on the default problem sizes for 32 processors, as suggested by the Stanford researchers, except for Barnes. In fact, Barnes is the only application that required modification for correct execution under a software DSM; we eliminated

System Constant Name	Default Value
Number of processors	16
TLB size	128 entries
TLB fill service time	100 cycles
All interrupts	400 cycles
Page size	4K bytes
Total cache per processor	128K bytes
Write buffer size	4 entries
Write cache size (AURC)	4 entries
Cache line size	32 bytes
Memory setup time	10 cycles
Memory access time (after setup)	3 cycles/word
PCI setup time	10 cycles
PCI *burst* access time (after setup)	3 cycles/word
Network path width	8 bits (bidirectional)
Messaging overhead	200 cycles
Switch latency	4 cycles
Wire latency	2 cycles
List processing	6 cycles/element
Page twinning	5 cycles/word + memory accesses
Diff application and creation	7 cycles/word + memory accesses

Table 1: Default Values for System Parameters. 1 cycle = 10 ns.

its busy waiting synchronization.

TSP uses a branch-and-bound algorithm to find the minimum cost tour of 18 cities. Barnes simulates the interaction of a system of 4K bodies under the influence of gravitational forces for 4 time-steps, using the Barnes-Hut hierarchical N-body method. Radix is an integer radix sort kernel. The algorithm is iterative, performing one iteration per digit of the 1M keys. Water is a molecular dynamics simulation computing inter- and intra-molecule forces for a set of 512 water molecules. Interactions are computed using an $\mathcal{O}(n^2)$ algorithm. Ocean studies large-scale ocean movements based on eddy and boundary currents. We simulate a 258×258 ocean grid. Em3d [7] simulates electromagnetic wave propagation through 3D objects. We simulate 40064 electric and magnetic objects connected randomly, with a 10% probability that neighboring objects reside in different nodes. We simulate the interactions between objects for 6 iterations.

5 Experimental Results

In this section we evaluate the performance of TreadMarks and AURC running on our simulated network of workstations. Subsection 5.1 studies the performance implications of each of the different forms of overhead tolerance afforded by protocol controllers. Subsection 5.2 compares the performance of our overlapping protocols against two versions of AURC, one with and one without prefetching. Subsection 5.3 evaluates the impact of different assumptions about messaging overhead, network bandwidth, and memory latency and bandwidth.

5.1 Isolated and Combined Gains

Figures 5 to 10 show the performance of the various forms of overlapping for the applications in our suite running on 16 nodes, under TreadMarks. From left to right, the bars represent:

Base: The non-overlapping version of the TreadMarks protocol.

I: The version where a computation processor is only used for complicated protocol processing, such as traversing lists and manipulating write notices. The more basic protocol actions are performed in software by its controller.

I+D: The version where diffs are computed using the hardware support we propose. Computation processors only run complicated protocol tasks, while diff generation and application are performed by the custom DMA engine.

P: The version where diff prefetching is applied to the standard TreadMarks protocol. All protocol processing, including diff-related operations, is performed by the computation processors.

I+P: The version where protocol controllers relieve their associated processors of the basic protocol tasks, and prefetching is applied. Diffs are generated and applied in software by the protocol controllers.

I+P+D: The version where computation processors do not perform basic protocol tasks, and prefetching and hardware-supported diffs are combined.

Note that versions Base and P do not assume protocol controllers, they represent the standard version of TreadMarks and TreadMarks with diff prefetching, respectively.

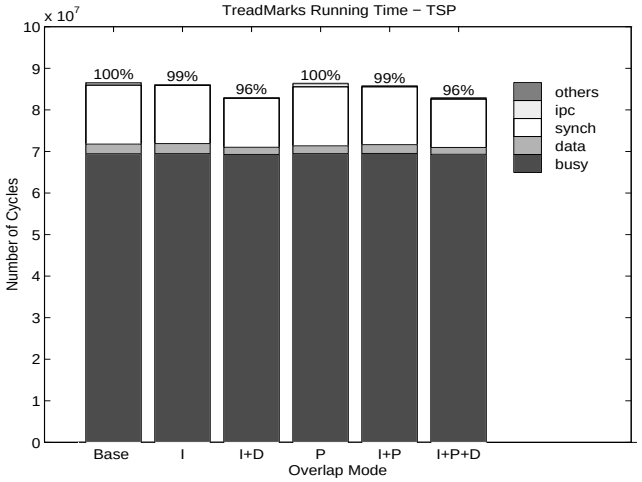

Figure 5: Performance of Overlapping Techniques for TSP under TreadMarks DSM.

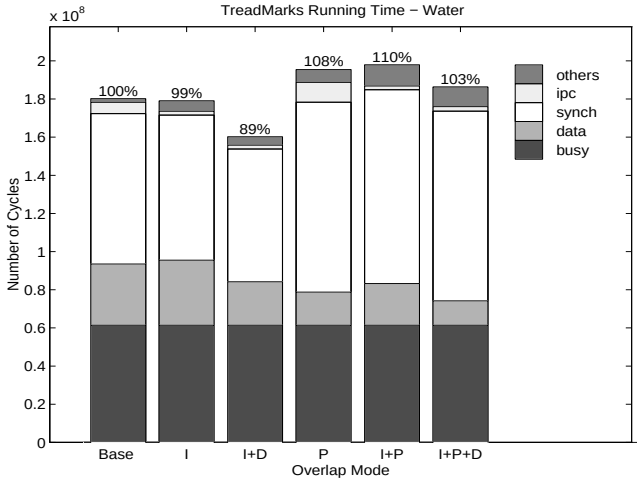

Figure 6: Performance of Overlapping Techniques for Water under TreadMarks DSM.

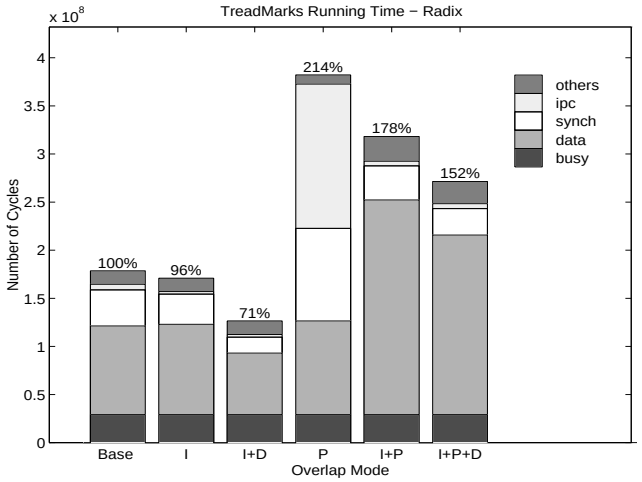

Figure 7: Performance of Overlapping Techniques for Radix under TreadMarks DSM.

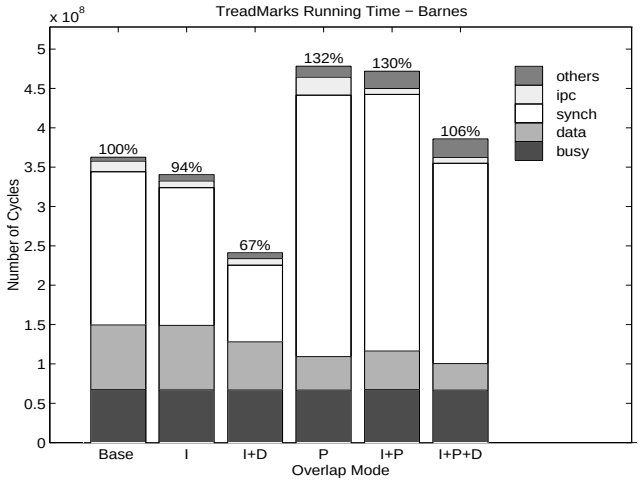

Figure 8: Performance of Overlapping Techniques for Barnes under TreadMarks DSM.

The bars in the figures show execution time once again broken down into busy time, page fetch latency, synchronization time, IPC overhead, and other overheads.

The Base execution time results for TreadMarks (the same as in section 2) show the great influence of data fetch and synchronization latencies in the applications we study. IPC overheads are not as significant, but can account for up to about 10% of the execution time as in Em3d. The cost of twinning, and diff generation and application contributes to both IPC and data fetch overheads and is often significant for our applications; these costs account for 1.5, 7.6, 20.6, 10.4, 26.7, and 20.9% of the execution time of TSP, Water, Radix, Barnes, Em3d, and Ocean, respectively. The techniques we study usually reduce these overheads. We now discuss the performance implications of each of the techniques in turn.

First, we consider the performance impact of moving basic protocol processing away from computation processors. This strategy affects the IPC time primarily. The I results show that this technique alone only provides non-negligible performance improvements for Em3d and Barnes. These gains come mainly from almost eliminating IPC overheads (Em3d) and reducing synchronization time (Barnes). Synchronization overhead is reduced because a lock holder (in lock synchronization) or a processor lagging behind the others (in barrier synchronization) can progress without much interference from other processors.

Adding our hardware support for diffs to protocol controllers achieves very good performance for all applications. The overall performance improvements achieved by I+D with respect to Base range from 4 to 39%. The more significant performance improvements provided by this technique come from reductions in data access and synchronization overheads. Data access improvements are a direct consequence of having hardware support for generating and applying diffs and, more importantly, avoiding twin creation altogether. The time required by these diff-related operations is reduced in 50, 44, 66, 44, 71, and 60% for TSP, Water, Radix, Barnes, Em3d, and Ocean, respectively, with respect to our Base re-

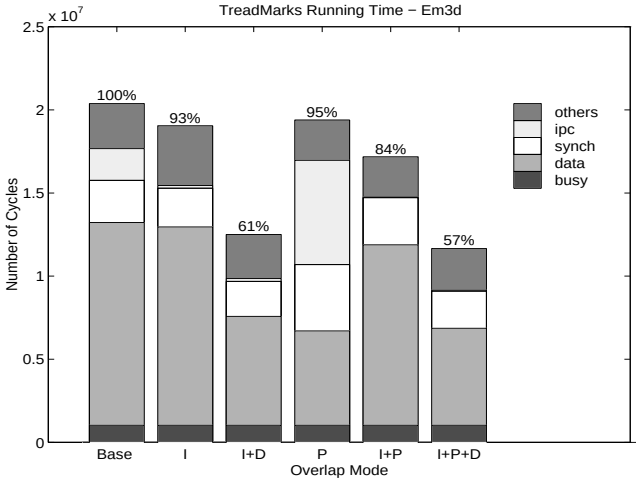

Figure 9: Performance of Overlapping Techniques for Em3d under TreadMarks DSM.

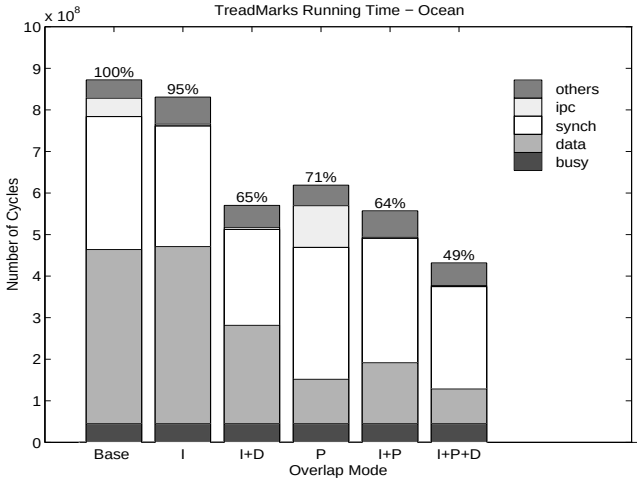

Figure 10: Performance of Overlapping Techniques for Ocean under TreadMarks DSM.

sults. In all but two instances (Ocean and Em3d), I+D outperforms all other overlapping techniques.

Now we turn to diff prefetching. Our experiments show that there are enough computation cycles to hide most or all of the latency of prefetches in our applications; the average time between a prefetch point and the actual use of the prefetched page ranges between 5K cycles for Em3d to 600K cycles for Ocean. However, prefetching alone only improves performance with respect to Base results for Em3d and Ocean. Even though prefetching does reduce page access latencies for all applications except Radix, it almost invariably causes a sizable increase in synchronization latency and may also generate significantly higher IPC overheads. Synchronization times increase because prefetching makes short critical sections extremely expensive, as in the case of Barnes and Water. IPC times increase as a result of prefetching when nodes guess their future access patterns incorrectly and end up prefetching pages they will not actually use. Each useless prefetch causes node interference (in the form of IPC time) that would not occur in Base. A high percentage of useless prefetches is a serious problem for Water and Radix: more than 85% of all their prefetches is useless. Another potential problem with prefetching is that it clusters page and/or diff transfers in time, which may degrade network, bus, and protocol controller performance significantly. Radix, Barnes, and Em3d are the applications that suffer the most from this type of degradation; network performance for these applications degrades by more than a factor of 2 under prefetching.

The combination of prefetching and I overlapping performs as well or better than prefetching in isolation in all cases, as a result of the virtual elimination of IPC overheads and reductions in synchronization time. Note that data access latencies often increase when we add I overlapping to diff prefetching. The reason for this result is that we assign low priorities to prefetches, making them wait for other more urgent contemporaneous commands to be executed by our protocol controllers.

Combining all the overlapping techniques afforded by our protocol controllers leads to the best running time results for the situations where I+D is not ideal.

Overall, our results show that our hardware-supported diffs are very efficient. The prefetching results indicate that this technique should not be used for the standard version of TreadMarks. For some applications prefetching performs well when combined with I overlapping as provided by our protocol controller. Independently of any hardware support, a less aggressive or adaptive prefetching strategy might reduce overheads, but it is not clear what this strategy should be. A complete analysis of different prefetching strategies is an interesting research topic, but is beyond the scope of this paper. In another paper [3] we propose and evaluate several diff prefetching strategies.

5.2 Comparison with AURC

In this section we compare the overlapping TreadMarks performance obtained in the previous section to AURC and AURC with prefetching. Figures 11 and 12 present these results. In each graph we plot, from left to right, the I+D version of the overlapping TreadMarks, AURC and AURC+P.

The results in the figures show that our prefetching strategy *never* improves the performance of AURC. For all applications prefetching reduces the page access latency exhibited by the protocol. However, these reductions are not large enough to outweigh the increased IPC and synchronization overheads entailed by prefetching [3]. There are two main reasons for this result: a) prefetching clusters requests at synchronization points, leading to degraded network and bus performance during these periods; and b) servicing prefetch requests requires processor intervention, which degrades performance severely when useless page prefetches dominate. Note that prefetching hurts AURC much more than TreadMarks, since the automatic-update traffic of AURC ends up competing for the same bandwidth as the prefetch traffic.

[3]Barnes exhibits an interesting result as AURC+P also increased this application's busy time significantly. This increase is a consequence of the synchronization style used when building the octree at each phase of the algorithm.

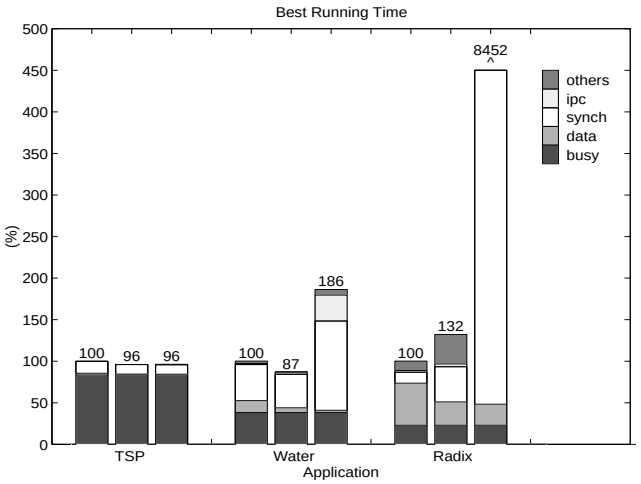

Figure 11: Comparison of Overlapping TreadMarks (left) with AURC (center) and AURC+P (right).

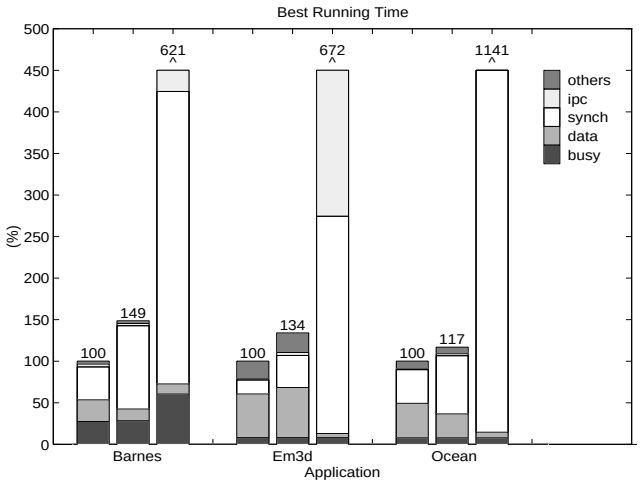

Figure 12: Comparison of Overlapping TreadMarks (left) with AURC (center) and AURC+P (right) Cont.

Our protocol controllers, on the other hand, have support for prefetching as commands can be assigned priorities. These priorities allow more urgent requests to be serviced ahead of prefetches.

Figures 11 and 12 also demonstrate that, in 4 cases (Radix, Barnes, Em3d, and Ocean), the overlapping TreadMarks outperforms the other DSMs. The performance advantage of overlapping TreadMarks over AURC ranges between 15 and 33%. These gains come mainly from significantly lower synchronization overheads that more than compensate for the higher data access latencies involved in our overlapping TreadMarks. AURC exhibits higher synchronization latencies for some applications due to excessive update traffic, which congests the network and, as a result, delays synchronization-related messages.

The performance of the overlapping TreadMarks and AURC is almost indistinguishable for TSP, while AURC performs 13% better than the overlapping TreadMarks for Water. It is also important to note that the overlapping version of TreadMarks performs at least as well as AURC for 5 out of our 6 applications, while the non-overlapping TreadMarks implementation is always outperformed by AURC.

5.3 Impact of Architectural Parameters

In order to understand the impact of different network and memory subsystems on DSM performance, in this section we evaluate the effect of messaging overhead (or the cycles spent on setting up the network interface), network bandwidth, and memory latency and bandwidth on our results. Figures 13 and 14 present a representative example of the impact of wide variations in messaging overhead and network bandwidth, respectively. We show results for the I+D version of our overlapping TreadMarks, along with the AURC running times. All running times are normalized to the overlapping TreadMarks results of the previous section. For reference, the network latency we have assumed so far translates into 2 microseconds, while the bandwidth corresponds to 50 MBytes/second. These settings can be considered aggressive for commercial networks nowadays, such as ATMs.

Figure 13 shows that the messaging overhead has little effect on the two DSMs. However, the results in this figure optimistically assume that update messages in AURC have a messaging overhead of a single cycle. When this assumption is eliminated, i.e. all messages in AURC suffer the same non-trivial messaging overhead, the performance of AURC starts degrading significantly as messages become more expensive. Figure 14 demonstrates that network bandwidth variations have a much greater impact on AURC than on TreadMarks. In fact, a bandwidth of 200 MBytes/second is required in this case for AURC to approach the performance of the overlapping TreadMarks implementation. Unfortunately, this level of bandwidth cannot currently be matched by low-cost, commercial networks.

Figures 15 and 16 present an evaluation of the impact of memory latency and bandwidth on overall performance. Again, we show results for the I+D version of our overlapping TreadMarks implementation and AURC. Running times are normalized to TreadMarks results. Our default memory latency has been 100 nanoseconds, while the default bandwidth has been 103 MBytes/second for cache block transfers.

Figures 15 and 16 show that memory performance affects the overlapping TreadMarks implementation more heavily than AURC. Memory latency has little effect on AURC, while the overlapping TreadMarks suffers severely with very high latency. Decreases in memory bandwidth affect TreadMarks slightly more severely than AURC.

6 Related Work

Several cache-coherent multiprocessors have been built in recent years [19, 1, 16]. These multiprocessors achieve very high performance at the cost of complex, hardware-intensive designs. Due to the complexity of the hardware, systems in this class may fail to keep up with microprocessor advances; i.e. the microprocessor chosen at the start of the project may end up being much slower than other off-the-shelf micropro-

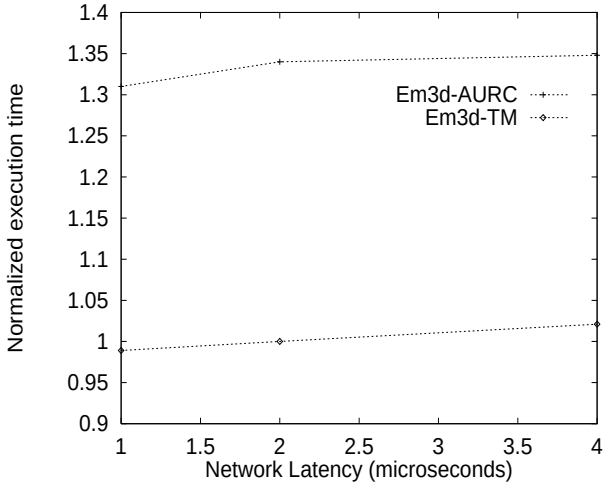

Figure 13: Effect of Messaging Overhead on Em3d Running Times. Bandwidth = 50 MB/sec.

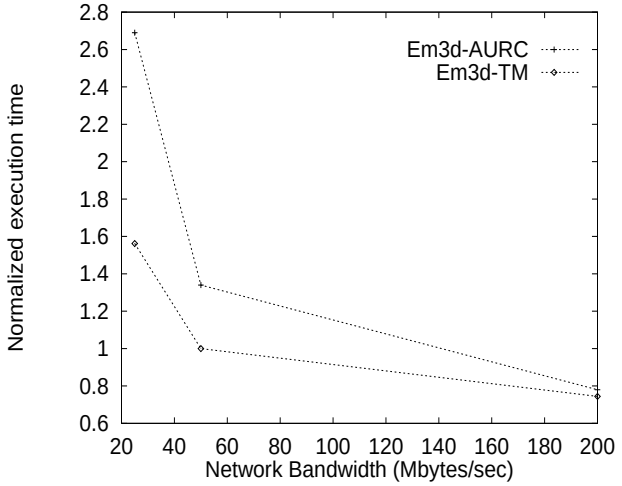

Figure 14: Effect of Network Bandwidth on Em3d Running Times. Latency = 2 microsecs.

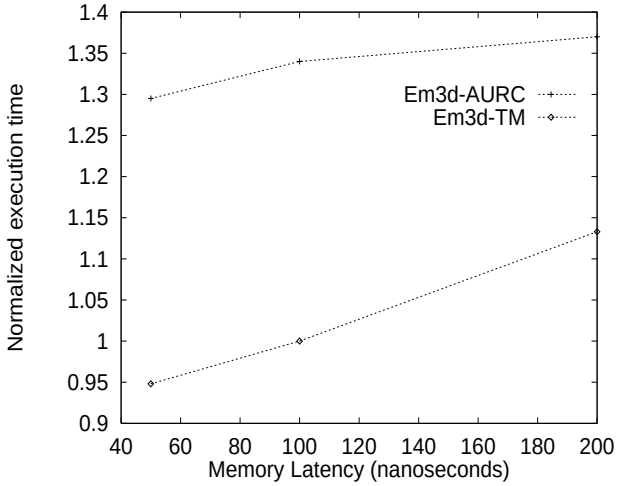

Figure 15: Effect of Memory Latency on Em3d Running Times. Bandwidth = 103 MB/sec.

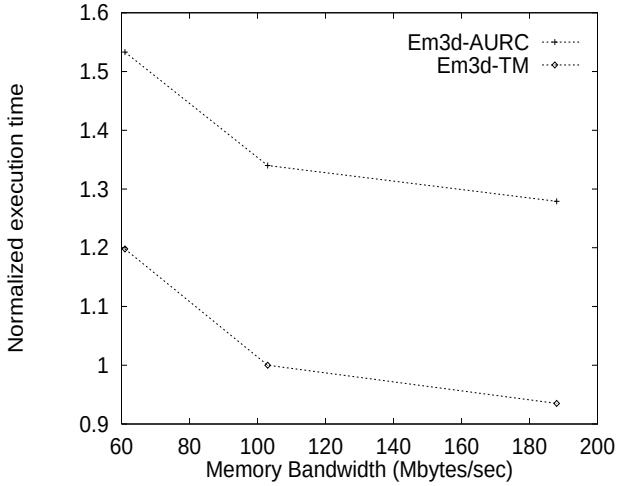

Figure 16: Effect of Memory Bandwidth on Em3d Running Times. Latency = 100 nsecs.

cessors by the time the machine is operational. Basing the machine's design on a next-generation microprocessor alleviates this problem, but may not solve it depending on how long it takes to build the machine. Due to its simplicity and reliance on commodity parts, our design leads to much lower costs and a much shorter design cycle, which places it in a different class of shared memory systems.

Our work borrows ideas from the research in programmable protocol processors being pioneered by the Stanford FLASH [18] and Wisconsin Typhoon [21] projects. These systems seek to provide flexible and efficient fine-gained sharing of data at the level of cache blocks. For such small coherence and transfer units, protocol processors must be extremely optimized to avoid becoming performance bottlenecks. More importantly however, short data transfers require very low network latency for efficient execution; these networks must then be backplanes tightly coupled with the rest of the node and protocol processor, which makes designs more complicated and expensive. In contrast to these approaches, our page-based coherence allows for a much simpler protocol processor and low-cost, commercial networks plugged onto a PCI bus.

Systems that attempt to provide fine-grained sharing of data in software but without as much custom hardware [23, 22] also require tight integration between the interconnection network and the rest of the node and place heavy (especially latency) demands on the network.

On the algorithmic side, our research builds on the work of a number of systems that provide shared memory and coherence in software using variants of release consistency. Both Munin [6] and TreadMarks [15] are designed to run on networks of workstations, with no hardware support for overlapping computation and communication. Our work shows that significant performance improvements can be accrued with hardware support for hiding communication latency and coherence overhead in such systems.

Prefetching for software DSMs has received little atten-

tion so far [11, 13, 10, 3]. In [11], Dwarkadas *et al.* describe a variation of TreadMarks (the Lazy Hybrid protocol) that does not use prefetching per se, but attempts to achieve the same result by piggybacking updates on a lock grant message when the last releaser of the lock has up-to-date data to provide and knows that the acquirer caches the data. Using this type of updates instead of our prefetches has the potential to reduce the number of messages exchanged by the system, however our more general prefetching strategy exhibits a greater potential to reduce data access latencies. In other work, Dwarkadas *et al.* [10] combine compile-time analysis with runtime support for prefetching. This strategy delivers good results as static analysis can predict the memory access behavior of loops. However, the strategy requires relatively complex compilers that may sometimes lack enough static information to perform the access analysis effectively. In contrast with this strategy and similarly to [13], our approach is solely based on dynamic, past-history information.

Previous research on prefetching for hardware-coherent multiprocessors has been extensive, e.g. [20, 8, 5, 2]. Our prefetching strategy differs from the above, in that it associates prefetching with synchronization and invalidation events. Furthermore our prefetching strategy fetches data in memory as opposed to the cache, and can still be profitably combined with the above strategies to reduce the cost of misses to the local memory.

Iftode *et al.* [12] proposes AURC and compares it against standard TreadMarks utilizing some of the same applications as we used. They show that AURC consistently outperforms TreadMarks which is consistent with our results. However, the application speedups they present in the paper are superior to the ones we found in our Base experiments. The main reason for this difference is that simulation time limitations prevented us from using inputs as large as theirs.

Kontothanassis and Scott [17] propose the Cashmere DSMs, which require the same type of hardware support as AURC. We believe that our claims and results regarding AURC apply to the Cashmere protocols as well.

7 Summary and Conclusions

In this paper we addressed the performance implications of using simple protocol controllers for hiding communication and coherence-induced overheads in software DSMs. Our protocol controllers provide three types of overhead tolerance: a) moving basic protocol tasks away from computation processors; b) prefetching of diffs; and c) generating and applying diffs with hardware assistance. We have developed overlapping versions of TreadMarks as they would run on a machine with protocol controllers like the ones we propose.

Using detailed execution-driven simulations of real applications, we isolated the impact of each of the overlapping techniques we studied and showed that the greatest performance benefits can be achieved by generating and applying diffs with hardware assistance. Reducing the burden of communication and coherence transactions on the computation processor is also beneficial but to a smaller extent. Prefetching was found to hurt performance in some cases, particularly when other forms of overlapping were not used as in standard TreadMarks. We have shown that these techniques can improve performance by up to about 50% with respect to standard TreadMarks.

We also studied two automatic-update-based DSMs, AURC and AURC using prefetching. Surprisingly, prefetching was found to always degrade the performance of AURC significantly. The overlapping TreadMarks protocol was shown to perform as well or better than AURC for 5 out of 6 applications.

We conclude that software DSMs supported by fairly simple hardware provide a low-cost approach to boosting the performance of parallel applications. However, further work on improving the performance of software DSMs is required, as significant amounts of overhead are still present in the applications we studied. Research on other overhead-tolerance techniques and on optimizing applications for software DSMs seems imperative. We have been pursuing these and other research avenues in the context of the NCP_2 project at COPPE/UFRJ.

Acknowledgements

We would like to thank the other members of the NCP_2 project, Gabriel Silva, Malena Hor-Meyll, Lauro Whately, and Julio Barros Jr., for their support and comments on all aspects of our project. We would also like to thank Luiz André Barroso and Czarek Dubnicki, who gave us useful comments on an early version of this paper. Last but not least, our thanks to John Bennett and the anonymous referees for helping improve this paper significantly.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[2] R. Bianchini and B.-H. Lim. Evaluating the Performance of Multithreading and Prefetching in Shared-Memory Multiprocessors. *To appear in Journal of Parallel and Distributed Computing, special issue on Multithreading for Multiprocessors*, October 1996.

[3] R. Bianchini, R. Pinto, and C. L. Amorim. Page Fault Behavior and Prefetching in Software DSMs. Technical Report ES-401/96, COPPE Systems Engineering, Federal University of Rio de Janeiro, July 1996.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991.

[7] D. Culler *et al.* Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[8] F. Dahlgren and P. Stenstrom. Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, January 1995.

[9] F. Dahlgren and P. Stenstrom. Reducing the Write Traffic for a Hybrid Cache Protocol. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[10] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[11] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20nd Annual International Symposium on Computer Architecture*, May 1993.

[12] L. Iftode, C. Dubnicki, E. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[13] P. Keleher. Coherence as an Abstract Type. Technical Report CS-TR-3544, Department of Computer Science, University of Maryland, Oct 1995.

[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 USENIX Winter '94 Technical Conference*, pages 17–21, Jan 1994.

[16] Kendall Square Research. *KSR1 Principles of Operation*, 1992.

[17] L. I. Kontothanassis and M. L. Scott. Distributed Shared Memory for New Generation Networks. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[18] J. Kuskin *et al.* The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[19] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[20] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[21] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[22] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[23] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain Access Control for Distributed Shared Memory. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, October 1994.

[24] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1994.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, May 1995.