The Affinity Entry Consistency Protocol *

Cristiana B. Seidel[†‡], R. Bianchini[†], and Claudio L. Amorim[†]

[†]COPPE Systems Engineering
Federal Univ. of Rio de Janeiro (UFRJ)

[‡]Dept. of Systems Engineering
State Univ. of Rio de Janeiro (UERJ)

{seidel,ricardo,amorim}@cos.ufrj.br

Abstract

In this paper we propose a novel software-only distributed shared-memory system (SW-DSM), the Affinity Entry Consistency (AEC) protocol. The protocol is based on Entry Consistency but, unlike previous approaches, does not require the explicit association of shared data to synchronization variables, uses the page as its coherence unit, and generates the set of modifications (in the form of diffs) made to shared pages eagerly. The AEC protocol hides the overhead of generating and applying diffs behind synchronization delays, and uses a novel technique, Lock Acquirer Prediction (LAP), to tolerate the overhead of transferring diffs through the network. LAP attempts to predict the next acquirer of a lock at the time of the release, so that the acquirer can be updated even before requesting ownership of the lock.

Using execution-driven simulation of real applications, we show that LAP performs very well under AEC; LAP predictions are within the 80-97% range of accuracy. Our results also show that LAP improves performance by 7-28% for our applications. In addition, we find that most of the diff creation overhead in the AEC protocol can usually be overlapped with synchronization latencies. A comparison against simulated TreadMarks shows that AEC outperforms TreadMarks by as much as 47%. We conclude that LAP is a useful technique for improving the performance of update-based SW-DSMs, while AEC is an efficient implementation of the Entry Consistency model.

1 Introduction

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

SW-DSMs based on relaxed consistency models can reduce these overheads by delaying and/or restricting communication and coherence transactions as much as possible. The Munin system [5], for instance, delays the creation and transfer of diffs (encoded modifications to shared pages) until lock release operations, so that messages can be coalesced and the negative impact of false sharing alleviated. TreadMarks [2] delays the coherence transactions even further, until the next lock acquire operation. The Midway [3] system also delays communication and coherence operations until a lock acquire transaction, but restricts these operations to the data that are associated with the lock.

Although effective at improving the performance of SW-DSMs, these protocols still involve a substantial amount of overhead: In Munin, updates are propagated to *all* processors sharing data modified within critical sections; on a page fault in TreadMarks, the faulting processor has to wait for diffs to be computed, received, and applied to the page before proceeding with its computation; in Midway, the processor acquiring a lock can only resume execution after the data associated with the lock become consistent locally.

Our work is based on the observation that all of these sources of overhead can be alleviated by dynamically predicting the lock acquisition order, and generating diffs away from the critical path of the processors that need them. In this paper we propose a novel SW-DSM, the Affinity Entry Consistency (AEC) protocol. The protocol hides the overhead of generating and applying diffs behind three types of synchronization delays: coherence processing at lock and barrier managers, waiting for a lock to become available, or waiting for all processors to reach a barrier.

In addition, the AEC protocol uses a novel technique, Lock Acquirer Prediction (LAP), for tolerating the overhead of transferring diffs through the network in Single Program Multiple Data (SPMD) applications. LAP attempts to predict the next acquirer of a lock at the time of the release, so that the acquirer can be updated even before requesting ownership of the lock. When the lock releaser makes a correct prediction, the protocol overlaps the communication and coherence overheads on the releaser with useful computation on the acquirer. In case of an incorrect prediction, the overlapped updates are wasted and the overhead of bringing data to the lock acquirer is exposed. When successful, LAP can also reduce the synchronization overhead in the presence of lock contention, since it effectively reduces the duration of critical sections and therefore the amount of time locks are held. LAP may be applied in other update-based SW-DSMs. In release-consistent systems such as Munin, LAP can be used to restrict the update traffic, while in systems like Midway, it can be used to overlap communication and computation.

*This research was supported by Brazilian FINEP/MCT, CNPq, and CAPES.

As in protocols based on Entry Consistency and its descendant Scope Consistency [9], in AEC a processor entering a critical section only receives the data associated with the section. In addition, the AEC protocol attempts to predict the lock acquisition order using LAP, hides coherence-related overheads, uses the page as its coherence unit, automatically associates data in a critical section with the lock that delimits it, and rarely requires program modifications to execute correctly.

Using execution-driven simulation of real applications, we show that LAP performs very well under AEC; LAP predictions are within the 80-97% range of accuracy. Our results also show that LAP reduces the amount of time devoted to memory access faults by as much as 62%. Overall, LAP improves performance by 7-28% for our applications. In addition, we find that most of the diff creation overhead in the AEC protocol can usually be overlapped with synchronization latencies. A comparison against simulated TreadMarks shows that AEC outperforms TreadMarks by as much as 47%. Most of this performance improvement is a result of lower data access and synchronization overheads. We conclude that LAP is a useful technique for improving the performance of update-based SW-DSMs, while AEC is an efficient implementation of the Entry Consistency model.

The remainder of this paper is organized as follows. The next section describes the three basic prediction techniques that comprise the LAP technique. Section 3 describes the AEC protocol in detail. Section 4 presents our methodology, application workload, and an overview of TreadMarks. In section 5 we present the results of our evaluation of LAP and AEC. We relate our work to previous contributions in section 6. Finally, in section 7, we present our conclusions and proposals for future improvements to AEC.

2 Predicting the Lock Acquisition Order

Our Lock Acquirer Prediction (LAP) technique attempts to predict the next acquirer of a lock based on three lower-level techniques: *waiting queue*, *virtual queue*, and *lock transfer affinity*. The next subsection describes these techniques in detail. Subsection 2.2 describes how these low-level techniques are combined to produce our LAP strategy.

2.1 Low-Level Techniques

Our first and simplest low-level technique, waiting queue, relies on the fact that the FIFO queue of processors waiting for access to a certain lock is a perfect description of the lock's acquisition order. So, if there is contention for a lock, the first processor on the FIFO queue will be the next acquirer of the lock.

However, a waiting queue for a lock might not exist at the time of the lock release, if there is only light competition for the lock. Thus, the idea behind our virtual queue technique is to create the waiting queue in advance of the processors actually requesting acquisition of the lock. We implement this strategy by inserting the transfer of *lock acquire notices* to the lock manager in the source code of applications. A virtual waiting queue is then created with the notices sent by the processors that intend to grab the lock in the near future. The next acquirer of the lock will likely be among the first few processors in the virtual queue.

Note that the burden of implementing virtual queues need not be placed on the programmer; a compiler can easily insert lock acquire notices in the applications. In this study we insert the notices manually to demonstrate that they are useful and therefore to motivate their implementation in a compiler.

The previous strategies may fail to produce a potential next lock acquirer, so we also define the lock transfer affinity technique. This technique is based on the fact that, for some SPMD applications, the next acquirer of a lock L_k released by processor P_i is frequently part of a small set of processors. Our lock transfer affinity technique uses the past history of lock ownership transfers to compute a set of potential next acquirers of the lock.

More specifically, we define the affinity $A_{ij}(k)$ of processor P_i for processor P_j, with respect to lock L_k, as the number of previous ownership transfers of L_k from P_i to P_j. In addition, we define the *affinity set* $S_i(k)$ of processor P_i with respect to lock L_k as the set of processors P_j with affinity $A_{ij}(k)$ 60% greater than the average affinity P_i has for other processors.[1]

2.2 The Lock Acquirer Prediction Technique

The LAP strategy combines the low-level techniques just described to compute the *update set* $U_i(k)$ that potentially contains the processor P_j, the next acquirer of lock L_k released by processor P_i. The size z of the update set is determined in advance by the user. The algorithm used to combine the techniques and compute $U_i(k)$ proceeds as follows:

1. If the waiting queue is not empty then $U_i(k) =$ the processor at the head of the queue; End.
2. Include the processors in $S_i(k)$ in $U_i(k)$;
3. If $U_i(k)$ is not complete (i.e. the number of processors in $U_i(k)$ is smaller than z), then include processors from the intersection of the virtual queue and the set of processors such that $A_{ij}(k) > 0$;
4. If $U_i(k)$ is still not complete, then insert processors from the virtual queue first and then the processors with $A_{ij} > 0$; End.

3 The AEC Protocol

In this section we describe our AEC protocol, which illustrates how an Entry Consistency-based protocol can take advantage of the LAP technique, while hiding the overhead of generating diffs behind synchronization delays.

3.1 Overview

The basic idea of AEC is to have a lock releaser send all modifications ever made inside the critical section protected by the lock

[1] Our threshold of 60% is admittedly arbitrary; we plan a study of the effect of varying this threshold for the near future.

to the processors in the update set (as determined by the lock's manager). These modifications are described by diffs, but diffs for the same page are merged into a single diff. The acquirer attempts to hide the overhead of applying these diffs and generating diffs for any pages modified outside of critical sections behind the synchronization overhead. Unfortunately, at the lock release point, the generation of diffs of the pages modified inside a critical section cannot be overlapped, in order to prevent the next lock acquirer from seeing potentially stale data.

Shared data modified outside of critical sections pose an interesting problem in this scheme, since the affinity concept does not apply to barriers. For this reason, shared data protected by barriers receive a different treatment than data protected by locks in AEC; barrier-protected data are kept coherent via invalidates. More specifically, diffs of barrier-protected data are generated eagerly, at the time of a lock acquire or a barrier arrival, and propagated to other processors only on access faults.

Thus, AEC attempts to hide the overhead of generating and applying diffs behind three types of delays:

- coherence processing at the lock and barrier managers;
- waiting for a lock to become available; or
- waiting for all processors to reach a barrier when there is load imbalance.

Figures 1 and 2 present a summary of the protocol actions involved in AEC, as a result of lock and barrier operations and on access faults to shared data. The next few subsections detail these actions.

3.2 Locks

On a lock acquire operation, the acquiring processor sends a message to the lock manager requesting ownership of the lock. Right after sending the request to the manager, the processor starts applying any diffs it has already received for being in the update set of a previous lock owner. Diffs are only applied to pages that are currently valid at the acquiring processor, so that only the pages that are likely to be used are updated; other diffs are saved for application at the time of an access fault to the corresponding page. The diffs are applied until they are exhausted or the manager's reply is received and the processor finds out who the last lock owner was. If all these diffs are applied before the receipt of the manager's reply, the acquiring processor starts generating the diffs corresponding to the shared data it modified outside of critical sections. Diffs are created until there are no more modified pages or the manager's reply is received. The twins used to generate these diffs are saved and all modified pages are write-protected. At this point, if the manager's reply has already arrived, the acquiring processor continues applying the diffs received, but only if the processor that sent them is indeed the last owner of the lock.

Upon receipt of the lock ownership request, the manager computes the new update set of the acquiring processor. In case the lock is currently taken by some other processor, the manager simply enqueues the id of the requesting processor at the end of a waiting queue for the lock. In case the lock is currently available, the manager replies to the acquiring processor's request with its new update set, the id of the last releaser, and a message informing whether the acquiring processor is in the update set of the last releaser. If the acquiring processor is not in the update set, the reply from the manager also includes a list of pages to invalidate.

Obviously, the lock acquire operation becomes much simpler when the last releaser of a lock is the acquiring processor itself, since there is no need to apply diffs, invalidate pages, or deal with a separate last owner.

On a lock release operation, the releasing processor generates diffs corresponding to modifications made inside the critical section it is about to leave, merges them with diffs received from the last owner of the lock, and sends the resulting diffs to its update set. After that, the releaser sends a message to the lock manager with a list of all pages represented in the merged diffs. This message also indicates that the releaser is giving up the ownership of the lock. Finally, the releasing processor unprotects all pages that had been modified outside the critical section and that were not modified inside it. Any diffs for these pages can be thrown away and their associated twins reutilized.

Note that the messages with the merged diffs must contain a lock acquire counter, so that the processors receiving different sets of diffs for the same lock can determine the most up-to-date one. This capability is important in the following scenario: suppose processor p holding a lock incorrectly guesses that processor q will be the next acquirer of the lock. Later, the actual next acquirer r correctly guesses q to be the next acquirer. Since processor q will receive two sets of merged diffs, in no particular order, it must be able to discard the outdated set.

3.3 Barriers

On a global synchronization event such as a barrier, each processor has to receive information on modifications performed inside and outside of critical sections. Our protocol implements barriers by dividing program execution into *steps*; a new step begins each time the processors depart from the barrier. Every processor has a step counter.

On the arrival at a barrier, each processor sends three lists to the barrier manager describing the step: a list of all lock variables it owned, a list of all pages accessed in the critical sections corresponding to these locks; and a list of all pages modified outside of critical sections. After these lists are sent to the barrier manager, the local processor starts to create the diffs corresponding to modifications made outside of critical sections. In effect, processors overlap the diff generation with barrier waiting time. However, in order to avoid generating diffs for pages that no other processor shares, a diff is only created for a page that was accessed by other processors in the previous barrier step and for which the local processor has received at least one request.

After the manager has received the messages from all processors, it determines, for each processor p, the set of processors to which p must send its diffs (corresponding to modifications made within critical sections) and/or notices that pages have been written outside of critical sections, the so-called *write notices*. A processor q is a candidate to receive diffs or write notices only if it has a valid copy of the page. A processor p sends a diff to processor q if p was the last owner of the lock under which the page was modi-

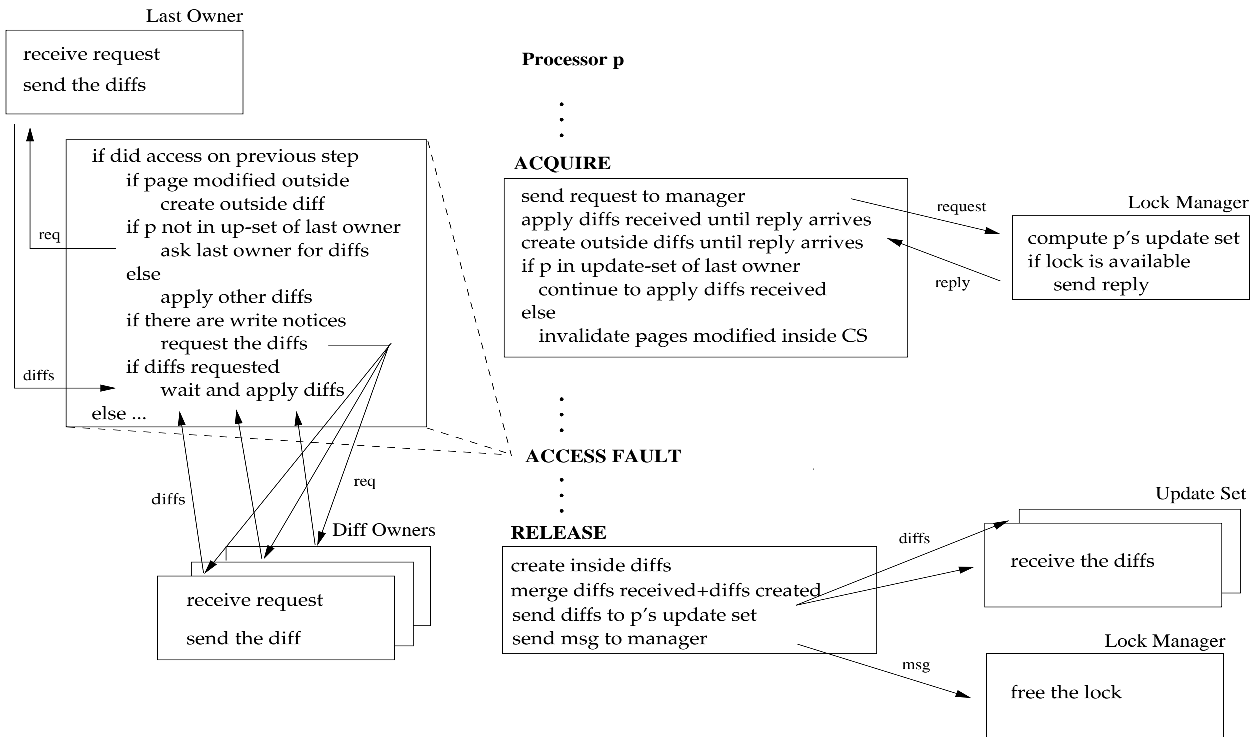

Figure 1: Actions Involved in Lock Operations and Access Faults Inside Critical Sections.

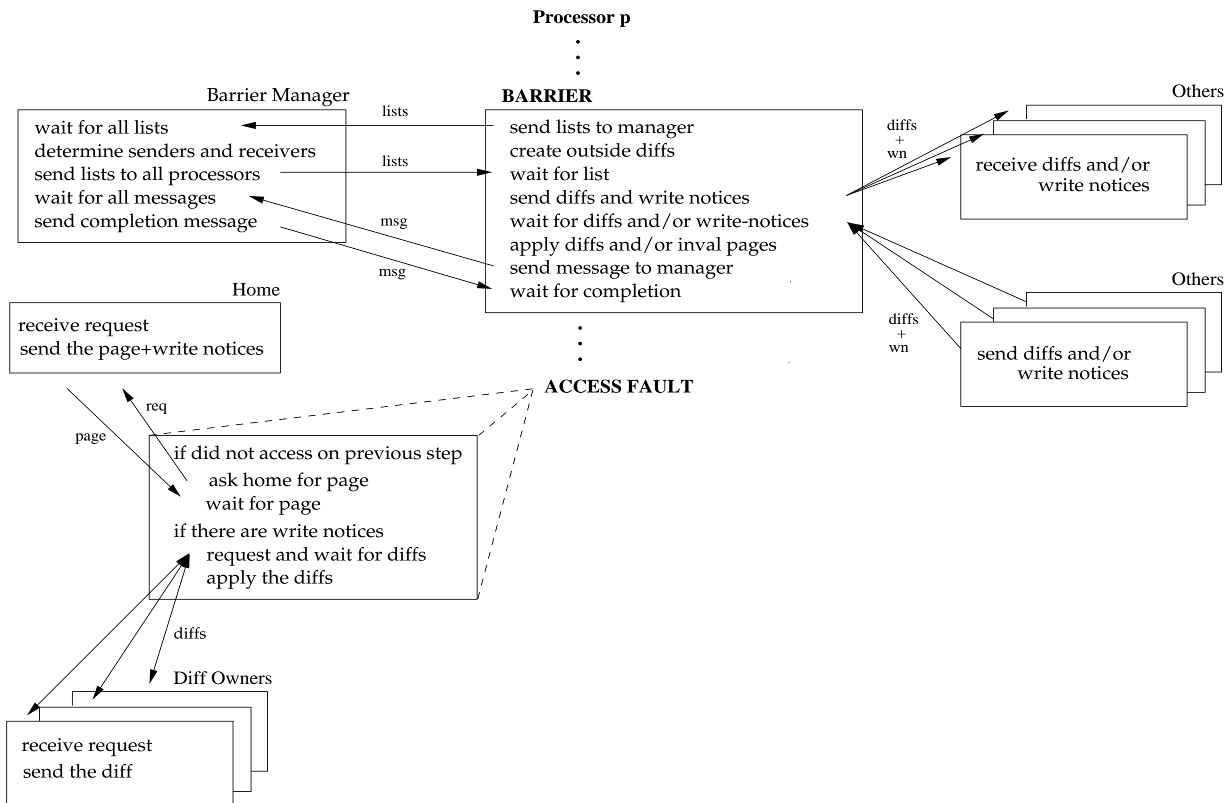

Figure 2: Actions Involved in Barrier Operations and Access Faults Outside Critical Sections.

fied. A processor p sends a write notice about a page to processor q if p modified the page outside of a critical section. In effect, this strategy avoids sending messages to processors that have not used the page, or that have already seen a write notice for it.

Since processors without valid copies of a page receive neither diffs nor write notices for it, they will need help at the time of an access fault to the page. The barrier manager, at the end of each step, chooses one of the processors with a valid copy of the page on arrival at the barrier to be the page's *home processor*; the processor that will help other processors reconstruct the page on access faults to it. The id of the home processor for each page is sent to all processors along with the set of processors they must communicate with.

After receiving these messages from the manager, the processors exchange diffs and write notices as specified by the manager. All diffs received are applied, while write notices cause the invalidation of the corresponding pages. After having completed these actions, processors communicate again with the manager, which can then determine the successful completion of the barrier event.

3.4 Access Faults

At the time of an access fault, the faulting processor must decide when it accessed the page last. If the processor did not access the page on the previous barrier step, it is not able to reconstruct the page independently, and must ask the page's current home node for help. In case the home node has a valid copy of the page, it sends it to the faulting processor. Otherwise, it sends the page and the write notices to be used by the requesting processor in bringing its copy of the page up-to-date. If the processor accessed the page on the previous step, the fault occurred as a result of one or more write notices received. In this case, the processor does not need to fetch a new copy of the page.

With a copy of the page in memory, the faulting processor can now collect the diffs it needs to validate the page. If the processor took the access fault while in a critical section and it was not in the update set of the last releaser, it asks the last releaser for the diffs it needs and applies them. If the faulting processor was in the update set of the releaser, it must apply any diff it previously received for the page but was unable to apply as it did not have a copy of the page at the time. Write faults inside of critical sections must be treated carefully, since the same page may have been modified outside of the critical section also. If a diff has not been created for the page before entering the critical section, the diff must be created and the corresponding twin eliminated.

For both inside and outside faults, if there are write notices (received during a barrier event in the past) for the page faulted on, the processor must then collect the necessary diffs according to the write notices.

4 Methodology and Workload

4.1 Multiprocessor Simulation

Our simulator consists of two parts: a front end, Mint [13], that simulates the execution of the processors and their registers, and a back end that simulates the SW-DSM protocol and the memory system in great detail. The front end calls the back end on every shared data reference. The back end decides which computation processors block waiting for memory (or other events) and which continue execution.

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 and private data accesses are assumed to take 1 cycle), local memory, an I/O bus, and a mesh network router (using wormhole routing). Network contention effects are modeled both at the source and destination of messages. Memory and I/O bus contention are fully-modeled. Table 1 summarizes the default parameters used in our simulations. All times are given in 10-ns processor cycles.

Constant Name	Default Value
Number of procs	16
TLB size	128 entries
TLB fill service time	100 cycles
All interrupts	4000 cycles
Page size	4K bytes
Total cache	256K bytes
Write buffer size	4 entries
Cache line size	32 bytes
Memory setup time	9 cycles
Memory access time	2.25 cycles/word
I/O bus setup time	12 cycles
I/O bus access time	3 cycles/word
Network path width	16 bits (bidir)
Messaging overhead	400 cycles
Switch latency	4 cycles
Wire latency	2 cycles
List processing	6 cycles/element
Page twinning	5 cycles/word + mem accesses
Diff appl/creation	7 cycles/word + mem accesses

Table 1: Defaults for System Params. 1 cycle = 10 ns.

Appls	# locks	# acq events	# barrier events
IS	1	80	21
Raytrace	18	3111	1
Water-ns	518	28128	33
FFT	1	16	7
Ocean	4	3328	900
Water-sp	6	533	33

Table 2: Synchronization events in our applications.

4.2 Workload

Our applications follow the SPMD model. Five of them come from the Splash-2 suite [14], while IS comes from Rice University. The applications exhibit widely different synchronization characteristics, as can be observed in table 2.

Appl	var #	# of lock events	% of total lock events	success rate			
				LAP	waitQ	waitQ+affinity	waitQ+virtualQ
IS	0	80	100.0%	92.0%	87.0%	92.0%	-
Raytrace	1	2049	65.9%	96.0%	96.0%	96.0%	-
	2-17	1046	33.6%	87.0%	3.4%	87.0%	-
Water-ns	4-515	27696	98.4%	80.4%	0.0%	66.0%	49.6%
FFT	0	16	100.0%	87.0%	87.0%	87.0%	-
Ocean	0	3200	96.2%	89.0%	78.0%	89.0%	-
Water-sp	0	240	47.2%	97.0%	95.0%	97.0%	-

Table 3: LAP Success Rates for z = 2.

IS uses bucket sort to rank an unsorted sequence of keys. In the first phase of the algorithm, each processor ranks its set of keys and updates a shared array with the rankings computed. In the second phase, each processor accesses the shared array to determine the final rankings of its keys. IS sorts 64K keys. The only lock in IS protects the shared array.

Raytrace renders a three-dimensional scene (*teapot*) using ray tracing. A ray is traced through each pixel in the image plane. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues (one per processor) are used with task stealing for load balancing. There is one lock to protect each task queue and one lock for memory management; the lock to assign an id to each ray has been eliminated, since id's are not really used.

Water-nsquared evaluates forces and potentials that occur over time in a system of water molecules. Water-nsquared uses an $\mathcal{O}(n^2)$ algorithm to compute these forces and potentials. The algorithm is run for 5 steps and the input size used is 512 molecules. Some locks accumulate global values but the majority of locks are used to update molecule forces. There is one lock for each water molecule.

FFT performs a complex 1-D FFT that is optimized to reduce interprocessor communication. The data set consists of 1M data points to be transformed, and another group of 1M points called roots of units. Each of these groups of points is organized as a 256 × 256 matrix. Locks are only used for initializing process ids.

Ocean studies large-scale ocean movements based on eddy and boundary currents. The input size we use is a 258 × 258 grid. Locks are used to identify processors and when processors compute local sums.

Water-spatial solves the same problem as Water-nsquared, but with an $\mathcal{O}(n \log n)$ algorithm. The algorithm is also run for 5 steps with 512 molecules. Locks are used for accessing only global values rather one lock per molecule.

Like under Scope Consistency, parallel applications might need slight modifications under AEC in two situations: a) when data are written within a critical section and are subsequently accessed without acquiring the corresponding lock; and b) when data are written outside of critical sections and must be visible before the next barrier. However, none of the applications in this study required modifications.

4.3 TreadMarks

In section 5 we compare the performance of our protocol against the one of TreadMarks. We chose to compare against TreadMarks instead of Midway, since, just like AEC, TreadMarks does not require the explicit association between shared data and synchronization variables. TreadMarks implements a lazy release consistency protocol in which the propagation of the modifications made to a shared page (diffs) is deferred until a processor suffers an access miss on the page. TreadMarks divides the program execution in intervals and computes a vector timestamp for each interval so that, on an acquire operation, it can determine the set of invalidations (write notices) the acquiring processor needs to receive. This vector describes a partial order between intervals of different processors. More details about TreadMarks can be found in [2].

5 Evaluating LAP and AEC

In this section we evaluate the performance of the LAP technique, the overlapping of diff generation and application with synchronization, and the AEC protocol.

5.1 Evaluating the LAP Technique

The lock managers in AEC compute update sets and maintain the waiting and virtual queues, and the affinity matrix A_{ij}. We evaluated LAP under AEC with the size z of the update set $U_i(k)$ varying from 1 to 3. Due to space limitations, however, we only present results for $z = 2$.

For each lock variable L_k, the lock manager computes the LAP success rate $H(k)$ as follows:

$$H(k) = \frac{no.\ times\ acquirer\ P_j\ was\ in\ U_i(k)\ of\ releaser\ P_i}{no.\ acquires\ executed\ on\ lock\ L_k}$$

In Table 3, we evaluate the contribution of each low-level prediction technique to the overall LAP success rate for the applications in our suite. The table shows, for each application and lock variable, the number of lock acquire events the variable is responsible for, the percentage of this number of events with respect to the total number of lock acquires, and the LAP success rates for each of its low-level techniques. The column labeled `LAP` presents $H(k)$, which combines the waiting queue, virtual

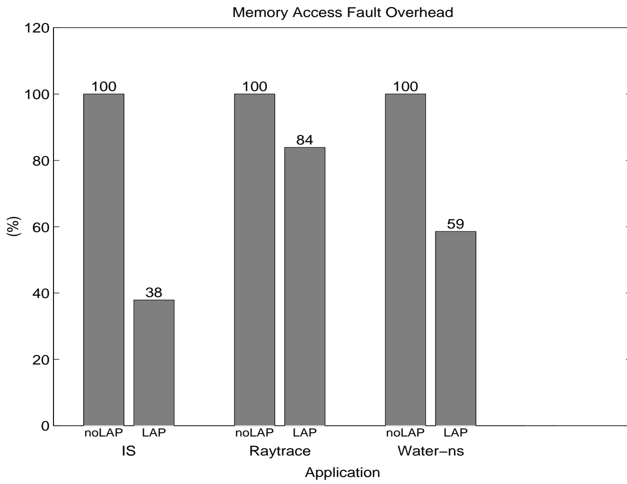

Figure 3: Access Fault Overheads Under AEC without LAP (noLAP) and AEC (LAP).

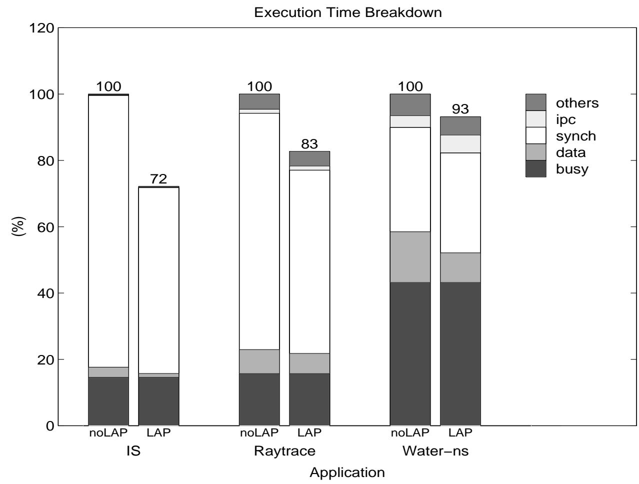

Figure 4: Running Time Under AEC without LAP (noLAP) and AEC (LAP).

queue, and affinity techniques; column `waitQ` presents the success rate of the waiting queue technique when applied in isolation; column `waitQ+affinity` presents the success rate of the combination of the affinity and waiting queue techniques; column `waitQ+virtualQ` presents the success rate of the combination of the waiting and virtual queue techniques[2].

Note that, due to the large number of lock variables in some of our applications, our table only presents data for variables with a reasonably high percentage of events. In addition, the table groups the variables that are logically related in the applications. A group's success rate is computed as the average success rate of the group variables weighted by the number of events of each variable. In Raytrace, all variables that protect the different task queues are put in a single group (var2-17). Water-nsquared, variables 4-515 protect the water molecules' structure.

As can be seen in the table, the LAP success rate is high, varying between 80% and 97% for the more important lock variables in our applications. For IS, one of the lock variables of Raytrace, FFT, Water-spatial, and Ocean, the waiting queue is the most effective technique. The virtual queue technique has a significant impact on the LAP success rate of Water-nsquared. The lock transfer affinity technique is effective for most lock variables of Raytrace, Water-nsquared, and Ocean.

In order to investigate the robustness of our results, we experimented with values of z in the range 1-3 and found that, increasing z from 1 to 2, increases the LAP success rate significantly. However, further increasing z to 3 improves the accuracy of LAP by very little; no more than 10%. Since a larger update set means that more data must be transferred through the network, $z = 2$ seems to be the best size.

In addition, we compared the LAP results taken from our simulated AEC to similar simulation-based implementations of the technique in TreadMarks and in a locally-developed release-consistent SW-DSM. Our results show that the LAP success rate does not vary significantly for applications with a large number of synchronization events per lock variable, even though the timing and ordering of these events (and therefore the outcome of the low-level techniques that comprise LAP) do change under the the different DSMs. Comparing LAP under AEC and TreadMarks, for instance, we find that success rates do not vary by more than 10% for our lock-intensive applications. This result demonstrates the robustness of the LAP technique.

[2] Virtual queues were only implemented when the success rate of the waiting queue technique was not high enough (less than 85%) and when the variable had a significant number of lock acquire events.

5.2 Evaluating LAP Under AEC

Since we apply LAP in our implementation of AEC to reduce the number of times processors are required to fetch remote diffs, it should have a direct effect on the overhead of memory access faults as observed by each processor. Thus, in figure 3, we evaluate the effectiveness of LAP under AEC, in terms of the overhead of access faults under AEC without LAP (left bar) and AEC (right bar) on a simulated 16-processor system. In AEC without LAP the data modified inside of a critical section are not eagerly transferred from releasers to acquirers of the corresponding lock. Instead, these transfers are done lazily at access faults.

In figure 3 we only plot data for the applications for which most of the synchronization overhead comes from lock operations. The bars in the figures are normalized with respect to the AEC without LAP results. The figure shows that LAP reduces the overhead of access faults by as much as 62% for IS. The smallest improvement, 16%, is achieved for Raytrace, since in this application almost 80% of the access fault overhead comes from cold-start access faults and twin generation latencies. LAP does not address these two types of overhead.

Whether improvements in access fault overhead significantly influence the overall execution time depends on the relative importance of this type of overhead and on the (indirect) effect of these improvements on other types of overheads. More specifically, a reduction in access fault overhead inside a critical section

significantly influences the lock waiting time when the application suffers from heavy lock contention.

In figure 4 we plot the simulated execution time of the same applications under AEC without LAP (left) and AEC (right) running on 16 processors. The bars in the figures show execution time broken down into busy time (busy), memory access fault overhead (data), synchronization time (synch), IPC overhead (ipc), and other overheads (others). The latter category is comprised by TLB miss latency, write buffer stall time, interrupt time, and the most significant of them, cache miss latency. The busy time represents the amount of useful work performed by the processor. Data fetch latency is a combination of coherence processing time and network latencies involved in fetching data as a result of page faults. Synchronization time represents the delays involved in waiting at barriers and lock acquires/releases. IPC overhead accounts for the time spent servicing requests coming from remote processors.

A running time comparison between AEC and AEC without LAP shows that the LAP improvements in fault overheads for IS and Raytrace have a significant effect on overall performance; LAP improves the performance of IS by 28% and the performance of Raytrace by 17%. The main reason for these results is that these applications exhibit heavy lock contention (as seen in table 3), and thus any improvement in fault overhead entails a similar reduction in lock synchronization latency. For instance, in IS the 62% reduction in access fault overhead induces a 42% reduction in the length of the critical section, which in turn produces a 37% improvement in lock synchronization overhead.

In addition, IS barriers are more efficient in the presence of LAP, again as a side-effect of the shorter critical sections entailed by the technique. In this application the barrier event occurs right after the highly-contended critical section, which serializes the execution of all processors. Thus, any extra cycles spent in the section contribute to greater waiting times at the barrier. More specifically, AEC without LAP entails 9 M more cycles in critical section time per processor than AEC. These extra cycles and the 66 M extra cycles spent waiting to acquire the lock under AEC without LAP account for the difference in barrier performance between the two versions of our protocol.

The running time improvement achieved by LAP on Water-nsquared comes solely from a reduction in the access fault overhead, as this application exhibits virtually no lock contention.

5.3 Evaluating the Benefits of Hiding Diff-Related Overheads

The overlapping of diff generation and application with synchronization overheads is also an important technique in the AEC protocol. A barrier event usually provides a good opportunity for overlapping as a result of load imbalance, while a lock acquire operation can significantly hide overheads when the acquiring processor is blocked as a result of lock contention. Lock release operations do not allow for overlapping overheads.

Table 4 presents data on the size of diffs and the extent to which our protocol is able to hide their generation cost. The leftmost column of the table lists our applications, while the other columns present the average size of diffs in bytes (Size), the average size of the diffs resulting from merges at lock release points (Merged Size), the percentage of diffs that are merged (Merged), the overall cost (in cycles) per processor of generating diffs (Create), and the percentage of this cost that is hidden by AEC (Hidden).

Appl	Size	Merged Size	Merged	Create	Hidden
IS	6140	6102	94%	5M	1.7%
Raytrace	351	83	22%	23M	85.6%
Water-ns	2332	104	34%	116M	74.6%
FFT	5280	6	0.02%	15M	99.9%
Ocean	2964	11	0.06%	107M	99.3%
Water-sp	727	15	6%	12M	96.9%

Table 4: Diff statistics in AEC.

The table shows that diffs are generally large in our applications, except for Raytrace and Water-spatial. For these applications, processors are only responsible for relatively small chunks of data in each page, such as a small number of water molecules in Water-spatial. The merged diffs are almost always very short; the exception here is IS in which processors write the whole shared array inside critical sections. Merged diffs only represent a non-negligible percentage of the total number of diffs in the three applications where the synchronization overhead is dominated by lock operations: IS, Raytrace, and Water-nsquared. The time it takes to generate diffs may represent a large percentage of the overall running time of applications. The results in the table show that a significant percentage of the diff creation cost is hidden in all applications except IS. The reason for this result is that in IS the vast majority of diffs are created at lock release points, where diff creation cannot be overlapped with other overheads.

AEC also allows for hiding the overhead of applying diffs in applications with lock synchronization. However, only Water-nsquared benefits from this characteristic of AEC, as the protocol hides 13% of the program's diff application cost.

5.4 Comparing AEC vs. TreadMarks

In this section we compare the performance of AEC against that of TreadMarks. We compare performance against TreadMarks instead of Midway, since, differently from Midway, both AEC and TreadMarks do not require the explicit association between shared data and synchronization variables. In figures 5 and 6, we plot the simulated execution time of each of our applications under TreadMarks (left) and AEC (right) running on 16 processors. The bars in the figures are broken down into the same categories as in figure 4. These figures show that our protocol outperforms TreadMarks for all but one application. The performance differences in favor of AEC range from 4% for Ocean to 47% for Raytrace. AEC and TreadMarks perform virtually the same for Water-nsquared.

Most of the improvement achieved by AEC comes from great reductions in synchronization and access fault times. Regarding the data access time improvement, there are two reasons why AEC leads to lower overhead than TreadMarks: a) the most important

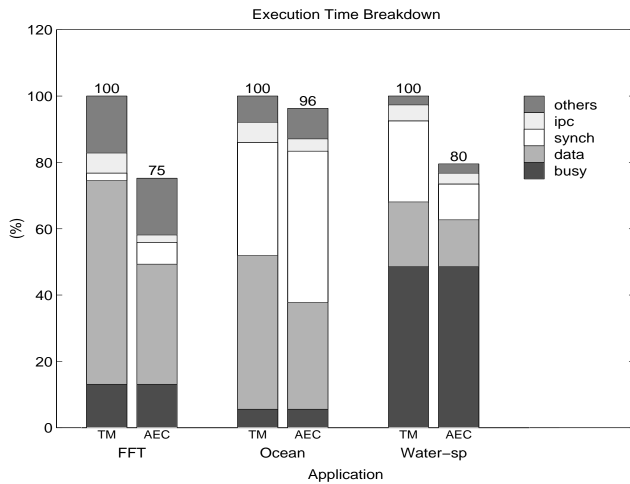

Figure 5: Execution Times Under TM and AEC.

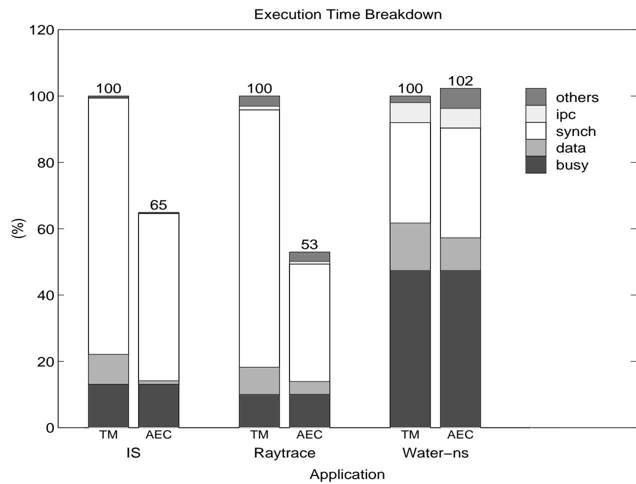

Figure 6: Execution Times Under TM and AEC.

one is that in TreadMarks diff creation is part of the critical path of both the generator and requester of the diff (thus, the diff creation overhead shows up under both `data` and `ipc` times in TreadMarks); and b) when LAP can correctly predict lock acquisition order, the lock acquirer does not take page faults within the critical section and thus need not fetch diffs from other processors.

For FFT, Ocean, Water-spatial, and Water-nsquared, the differences in access fault times are mostly a result of performing diff creation away from the critical path of processors. For IS, Raytrace, and to some extent Water-nsquared, these differences come from LAP's elimination of most page faults within the applications' critical sections.

The synchronization overhead of IS, Raytrace, and Water-spatial is much greater under TreadMarks than under AEC. IS, Raytrace, and Water-spatial exhibit more efficient lock synchronization under AEC again as a result of shorter critical sections and heavy lock contention. In addition, barriers are more efficient under AEC for IS and Water-spatial, again as a side-effect of the shorter critical sections entailed by LAP and the program structure with a barrier following a highly-contended critical section.

Synchronization overheads are not always smaller under AEC however, as seen in the Water-nsquared, FFT, and Ocean results. For all these applications the barrier performance is degraded under AEC, while the lock performance is roughly the same under the two protocols. For applications such as Water-nsquared, LAP generates load imbalance in the initial phases of computation, as the technique requires a certain number of lock transfers before it can predict them correctly. The problem is that during these phases some processors predict transfers correctly and others do not. The resulting load imbalance hurts the performance of barriers.

For FFT and Ocean, the performance of barriers is worse under AEC for a different reason. In these applications, AEC generates a large number of diffs during most barrier events, increasing the utilization of the local memory buses, which in turn degrades AEC's messaging performance. This is a serious problem for AEC, given that it requires more messages than TreadMarks at barrier events.

6 Related Work

Current SW-DSMs rely on relaxed memory consistency models that require the virtual shared memory to be consistent only at special synchronization events. In Release Consistency (RC) [7], operations on shared data are only guaranteed to be seen by all processors at lock release operations. Lazy Release Consistency (LRC) [10] further relaxes RC by delaying coherence actions until the next lock acquire operation. Entry consistency (EC) [3] proposes an even more relaxed model of memory consistency that explores the relationship between synchronization objects that protect critical sections and the shared data accessed within the sections. Like LRC, EC delays propagation of updates until the next acquire operation. Scope Consistency (ScC) [9] proposes a memory consistency model (and associated protocol) that attempts to achieve the advantages of EC without having to explicitly bind data to synchronization objects. The ADSM protocol [11] implements a variation of EC that does not require explicit bindings either. Both the ScC and AEC protocols assume update-based coherence protocols, while ADSM only uses updates for single-writer data protected by locks.

SW-DSM protocols vary mainly in the way they manage the propagation of coherence information at the synchronization points; Munin [5], TreadMarks [2] and its Lazy Hybrid variation [6], and Midway [3] are important examples. AEC leads to much less communication than in Munin, since updates are only sent to the update set of the lock releaser, as opposed to all processors that shared the modified data. Like AEC, TreadMarks and Midway seek to avoid communication, but expose all the overhead of generating, fetching, and applying diffs to bring pages up-to-date. The Lazy Hybrid protocol avoids the overhead of fetching diffs by piggybacking them 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. AEC tackles the cost of diff handling more aggressively than these systems, using overlapped diffs and the LAP technique. In addition, AEC provides a simpler programming interface for EC than Midway.

The AEC protocol is based on a memory consistency model

equivalent to ScC. However, AEC differs from the ScC protocol in two important ways. AEC is a software-only algorithm, whereas the ScC protocol, as evaluated in [8], uses automatic update hardware (even though an all-software implementation of ScC is also possible). In addition, AEC tackles several overheads of traditional SW-DSMs by overlapping coherence and communication operations with useful work or other overheads. AEC [1] and ScC were developed simultaneously and independently.

AURC [8] and the protocol controller-based DSMs in [4] also seek to overlap communication and coherence overheads with useful computation. Both of these approaches also require a small amount of custom hardware.

The optimization of critical sections has also been the object of study in the context of cache-coherent multiprocessors. Trancoso and Torrellas [12] have studied the use of data prefetching and forwarding to reduce the number of cache misses occurring inside of critical sections. Forwarding in this context is used to send the data modified by a lock holder to the first processor waiting at the lock's queue; no data is sent out when there is no lock contention. Thus, their strategy has a similar effect to our LAP technique, but only when processors contend for lock access.

7 Conclusions

In this paper we proposed the AEC protocol, which relies heavily on the novel LAP technique and on overlapping diff generation and application overheads. Our analysis of the LAP technique showed that LAP is very successful at correctly predicting the next acquirer of a lock, which leads to a significant performance benefit to AEC. Hiding the generation of diffs behind synchronization overheads also improves performance significantly. However, most of the overhead of applying diffs is still exposed in AEC. A comparison against TreadMarks shows that AEC outperforms TreadMarks for 5 out of our 6 applications, mostly as a result of lower data access and synchronization overheads.

In summary, our main contributions have been the proposal and evaluation of AEC and LAP. AEC has been shown an efficient SW-DSM protocol. LAP is a general technique for predicting the lock acquisition order and can potentially be used for optimizing other update-based SW-DSMs.

Acknowledgements

The authors would like to thank Leonidas Kontothanassis and Raquel Pinto for their help with our simulation infrastructure. We would also like to thank Liviu Iftode, who helped us with applications. Luis Favre suggested the virtual queues technique, we thank him also. Finally, Raquel Pinto, Paula Maciel, and Luis Monnerat suggested modifications that helped improve the text.

References

[1] C. L. Amorim, C. B. Seidel, and R. Bianchini. The Affinity Entry Consistency Protocol. Tech. Report ES-388/96, COPPE Systems Engineering, Federal University of Rio de Janeiro, May 1996.

[2] C. Amza, A. Cox, S. Dwarkadas, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), Feb 1996.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the IEEE COMPCON'93 Conference*, Feb 1993.

[4] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 13(3), Aug 1995.

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

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th International Symposium on Computer Architecture*, May 1990.

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

[9] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[10] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.

[11] L. R. Monnerat and R. Bianchini. ADSM: A Hybrid DSM Protocol that Efficiently Adapts to Sharing Patterns. Tech. Report ES-425/97, COPPE Systems Engineering, Federal University of Rio de Janeiro, March 1997.

[12] P. Trancoso and J. Torrellas. The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. In *Proc. of the 1996 International Conference on Parallel Processing*, Aug 1996.

[13] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proc. of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

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