A Comparison of Entry Consistency and Lazy Release Consistency Implementations

Sarita V. Adve, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, Willy Zwaenepoel
Departments of Computer Science and Electrical and Computer Engineering
Rice University
Houston, TX 77005-1892
{sarita,alc,sandhya,rrk,willy}@rice.edu

Abstract

This paper compares several implementations of *entry consistency* (EC) and *lazy release consistency* (LRC), two relaxed memory models in use with software distributed shared memory (DSM) systems. We use six applications in our study: SOR, Quicksort, Water, Barnes-Hut, IS, and 3D-FFT. For these applications, EC's requirement that all shared data be associated with a synchronization object leads to a fair amount of additional programming effort. We identify, in particular, extra synchronization, lock rebinding, and object granularity as sources of extra complexity. In terms of performance, for the set of applications and for the computing environment utilized neither model is consistently better than the other. For SOR and IS, execution times are about the same, but LRC is faster for Water (33%) and Barnes-Hut (41%) and EC is faster for Quicksort (14%) and 3D-FFT (10%).

Among the implementations of EC and LRC, we independently vary the method for *write trapping* and the method for *write collection*. Our goal is to separate implementation issues from any particular model. We consider write trapping by compiler instrumentation of the code and by twinning (comparing the current version of shared data with an older version). Write collection is done either by scanning *timestamps* or by building *diffs*, records of the changes to shared data. For write trapping in EC, twinning is faster if data is shared at the granularity of a single word. For larger granularities than a word, compiler instrumentation is faster. For write trapping in LRC, twinning gives the best performance for all applications. For write collection in EC, timestamping works best in applications dominated by migratory data, while for other data diffing works best. For LRC, increased communication overhead in transmitting timestamps becomes an additional factor working in favor of diffing for applications with fine-grain sharing.

This work is supported in part by the National Science Foundation under Grants CCR-91163343, CCR-9211004, CCR-9410457, CCR-9502500, and CDA-9502791, by the Texas Advanced Technology Program under Grants 0036404013 and 003604016, and by a grant from Tech-Sym, Inc.

1 Introduction

Distributed shared memory (DSM) enables processes on different machines to share memory, even though the machines physically do not share memory [11]. This approach is attractive since most programmers find shared memory easier to use than message passing, which requires them to explicitly partition data and manage communication. Here, we focus on software implementations of DSM. Early such systems suffered from high communication overhead. To combat these problems, software DSM implementations have turned to relaxed memory models [6]. Two popular models in use with current DSM systems are lazy release consistency (LRC) [8], used in TreadMarks [10], and entry consistency (EC) [3], used in Midway [13].

Both LRC and EC allow delaying the propagation of modifications to shared data until a synchronization operation occurs. To do so, both models require that the programmer use only system-provided synchronization primitives. EC, in addition, requires shared data to be associated with a synchronization object. This additional requirement complicates the programming model in EC compared to LRC. In both models, synchronization primitives are divided into releases and acquires, and consistency actions are taken at the time an acquire occurs. In EC only the data that is associated with the synchronization object being acquired is made consistent at the acquirer. LRC instead makes all shared data consistent at the acquirer.

This paper presents a detailed comparison of the two models. To make such a comparison, we must answer two methodological questions: 1) what is the proper program for a particular application in each model?, and 2) what is the proper implementation of each model? To answer the first question, we obtained a number of programs written by the authors of Midway and TreadMarks, representative systems for EC and LRC, respectively These programs are SOR, Quicksort, and Water. Additional applications were written in the same programming style for each model. We also explore some alternatives in Sections 3.3 and 7.2.

To address the second question, we have considered a number of implementations of each model, both previously published methods and improvements thereof. In the comparison between the models, we represent each by the implementation that performs the best for the particular application.

The implementations of the two models vary both the method of *write trapping* and the method of *write collection*. Write trapping detects what shared memory locations have been changed, and is done in one of two ways. The *twinning* mechanism maintains an unmodified copy of a shared data object (called a twin), and compares the current copy with the twin to determine the changes to shared data. Twinning requires no support from the compiler. In contrast, with *compiler instrumentation*, the compiler emits extra instructions that set software dirty bits when the corresponding shared data objects are changed.

Write collection refers to the mechanisms used for determining what modified data needs to be propagated to the acquirer. This requires a mechanism for recording at what "logical" time a given data item was last modified. Again, two methods are considered. The first method uses *timestamps*. A timestamp is associated with each shared data item, and records when that data item was last modified. The second method creates *diffs*. Diffs are associated with execution intervals and record the changes to a shared data item made during that execution interval.

Although in theory one could combine each of the two methods for write trapping with each of the two methods for write collection, only three out of the four combinations are explored in this paper. The combination of compiler instrumentation and diffing is not considered in any detail because its memory requirements appear prohibitive. The combination of compiler instrumentation and timestamps is used to implement EC in Midway [13]. Twinning and diffing is used to implement LRC in TreadMarks [10]. This paper attempts a more methodical exploration of the various combinations, and considers in addition to the above: compiler instrumentation and timestamps for LRC, twinning and timestamps both for LRC and EC, and a twinning and diffing algorithm for EC that improves over earlier published methods [13]. Table 1 summarizes the various implementations.

An implementation of EC or LRC must also decide whether to use an invalidate or an update protocol. Previous work [9] has shown that an invalidate protocol results in the best performance for LRC. An invalidate protocol is therefore used in TreadMarks, and also in the implementations of LRC in this study. In contrast, the papers on EC have argued that by restricting the regions of memory to be made consistent to those associated with a synchronization object, EC is best served by an update protocol [3]. Midway uses an update protocol, and so do the implementations of EC in this study. We will discuss the influence of these implementation decisions on the performance results in Section 7. We could not directly compare TreadMarks and Midway, because TreadMarks runs on Unix and Midway runs on Mach. Differences in communication and page fault handling overheads between Unix and Mach would obscure the differences between the two models.

Trapping / Collection	Model	Comp.Ins.	Twinning
Timestamping	EC LRC	~Midway new	new new
Diffing	EC LRC		[13] improved TreadMarks

Table 1 Combinations of Write Trapping and Write Collection Explored in This Paper

Our comparison of EC and LRC shows no clear winner in terms of performance for the environment and the applications examined. EC outperforms LRC if the data associated with a lock is larger than a page. If it is smaller than a page, then EC outperforms LRC if there is false sharing, while LRC outperforms EC if there is spatial locality resulting in a prefetch effect. With respect to write trapping, twinning is faster if the program requires sharing of individual words. On the other hand, if the smallest shared datum is larger than a word, compiler instrumentation is faster. With respect to write collection, for migratory data, timestamps perform better than diffs. For other types of data, the higher computation overhead and higher communication overhead due to sending timestamps yields poorer performance for timestamping. These overheads are more significant in LRC.

The experiments were carried out on a 100-Mbps point-to-point ATM network connecting 8 DECstation-5000/240s. The applications used in the comparison are: Red-Black Successive Over-Relaxation (SOR), Quicksort, Water, Barnes-Hut, Integer Sort (IS), and three-dimensional FFT (3D-FFT). SOR and Quicksort are small test programs. Water and Barnes-Hut are from the Splash suite [12]. IS and 3D-FFT are from the NAS benchmark suite [2].

The outline of the rest of this paper is as follows. Section 2 presents the applications used in this study. Section 3 discusses EC and LRC, contrasts programming in EC and LRC, and illustrates the differences with examples from the applications. Section 4 discusses write trapping. Section 5 discusses write collection. Section 6 describes the experimental environment. Section 7 compares the performance of EC and LRC. Section 8 compares the performance of the write trapping and write collection techniques, both for EC and LRC. Section 9 presents related work. Section 10 summarizes our conclusions.

2 Applications

We used six programs in this study: Red-Black SOR, Quicksort, Water, Barnes-Hut, Integer Sort, and 3D-FFT. SOR, Quicksort, and Water were used in earlier studies of Midway [13] and TreadMarks [10]. We describe the applications as written for a sequentially consistent system.

SOR uses Red-Black Successive Over-Relaxation to solve partial differential equations. The program determines the steady state values in a system where the boundary elements are kept constant. A matrix of floating-point numbers represents the system in which the four edges are kept constant. The program iterates over this matrix, computing a new value for each element based on its four neighbors. Each iteration is made up of two phases separated by a barrier: the black elements are updated based on the values for the red elements computed in the previous phase and vice versa. The matrix is divided into roughly equal size bands of consecutive rows, with each band being assigned to a different processor. Communication occurs across the boundary between bands.

Quicksort (QS) uses a centralized task-queue based approach to sort an array of integers. Initially, the entire array is inserted in the task queue. A processor repeatedly dequeues a sub-array to be sorted from the queue and recursively applies the quicksort algorithm to the dequeued element. Application of the algorithm results in partitioning the dequeued element into two sub-arrays around the chosen pivot. The smaller partition is enqueued in the task queue and the processor continues to work on the larger partition. When the partition size reaches a threshold of 1024 integers, the partition is sorted locally using a bubblesort algorithm.

Water, from the SPLASH suite [12], is a molecular dynamics simulation. The molecules are distributed equally among processors. There are two key phases in each timestep. In the first phase, called the force computation phase, a processor updates the forces due to the interaction of its molecules with those of half of the other processors. The force computation requires reading the displacement vectors of the interacting molecules, which are calculated in the previous timestep. In the second phase, called the displacement computation phase, a processor updates the displacements of its molecules based on the forces calculated in the previous phase. The phases are separated by barriers. A lock protects access to each molecule record during the force computation phase, because each force value is updated by several processors. No lock is required during the displacement computation phase, because each processor only updates the displacements of the molecules it owns. As suggested in the SPLASH report [12], in the force computation phase, each processor uses a local variable to accumulate its updates to a molecule's force record. At the end of the phase, the processor acquires a lock on each molecule that it needs to update and applies the accumulated updates at once.

Barnes-Hut is a simulation of a system of bodies influenced by gravitational forces. A body is represented as a point mass that exerts forces on all other bodies. The algorithm uses a hierarchical oct-tree representation of space in three dimensions. The space is broken into cells. The internal nodes of the oct-tree represent the cells, and the leaves represent the bodies in the corresponding cells. Each time step consists of the following key phases: a processor traverses the tree to obtain a set of bodies that results in good load balance between processors; it then computes the forces on these bodies; and finally it computes the new positions of the bodies. We refer to these phases as the load balancing phase, the force computation phase, and the position computation phase. Barriers separate the different phases. No locks are required since in each phase at most one processor updates any data item.

The **Integer Sort (IS)** NAS benchmark requires ranking an unsorted sequence of N keys. The *rank* of a key in a sequence is the index value i that the key would have if the sequence of keys were sorted. All the keys are integers in the range $[0, B_{max}]$ and the method used is counting or bucket sort. The amount of computation required for this benchmark is relatively small – linear in the size of the array N. The amount of communication is proportional to the size of the key range, since an array of size B_{max} has to be passed around between processors. The program consists of two phases. In the first phase, each processor first ranks its set of keys. It then requests exclusive access (via a lock) to a shared array, and increments the values in the shared array with its own rankings, keeping a local copy of the current values in the shared array. In this phase the shared array exhibits migratory behavior. A barrier ends the first phase. In the second phase each processor reads the final values in the shared array in order to determine the final ranks for its local keys.

The **3D-FFT** NAS benchmark numerically solves a partial differential equation using forward and inverse FFT's. Assuming an $n_1 \times n_2 \times n_3$ input array (say A) organized in row-major order, we distribute the array elements along the first dimension of A. That is, for any i, all elements of $A[i, *, *]$ are contained within a single processor. A 1-D FFT is first performed on the $n_1 \times n_2$ n_3-point vectors, and then on the $n_3 \times n_1$ n_2-point vectors. For these phases, each processor can work on its part of the array without any communication. A barrier separates these first two phases from the third and final phase, which is a transpose followed by a 1-D FFT on the $n_2 \times n_3$ n_1-point vectors. During the transpose, with n processors, each processor needs to read $1/n$ of its data from each of the other processors.

3 Entry Consistency vs. Lazy Release Consistency

Sections 3.1 and 3.2 summarize the two consistency models under discussion. For more extensive discussions we refer the reader to the papers introducing release consistency [7], lazy release consistency [8], and entry consistency [3]. Section 3.3 discusses the differences in programming under the two models.

3.1 Entry Consistency (EC)

In EC, all shared data must be explicitly declared as such in the program text, and associated with a synchronization object that protects access to that shared data. Processes must synchronize via system-supplied primitives. Synchronization operations are divided into acquires (getting access to shared data) and releases (granting access to shared data). After completing an acquire, EC guarantees that a process sees the most recent version of the data associated with the acquired synchronization variable. In our implementations of EC, synchronization primitives include exclusive locks, read-only locks, and barriers. Following the practice adopted in Midway, shared data is associated with locks but not with barriers.

3.2 Lazy Release Consistency (LRC)

Release consistency is similar to EC in that it guarantees consistency only after a synchronization operation. In the lazy version of release consistency (LRC), the propagation of consistency information is postponed until the time of an acquire, as in EC. Unlike EC, however, there is no notion of association between synchronization objects and data. This reduces programming effort, but it has the disadvantage that, at an acquire, LRC must make all shared data consistent between the releaser and the acquirer. The implementation of LRC used in this paper provides exclusive locks and barriers. There is no need for read-only locks for the application suite we consider.

3.3 Programming Under EC and LRC

In contrast to LRC, EC requires programmers to associate (or bind) every shared data object with a lock, and to access a shared data object only after acquiring the corresponding lock. The lock may be acquired in read-only or exclusive mode, as appropriate. We describe several scenarios where these requirements result in modifications to a program written for sequential consistency, and illustrate these with examples from our application suite in Section 2. No changes were required for the programs in our application suite to work correctly on LRC.

Barriers. We first look at programs in which different phases are separated by barriers. Consider a phase where each processor reads part of a data structure which is modified by other processors in a previous phase. With sequential consistency and LRC, the barrier at the beginning of the phase ensures that each processor reads the up-to-date value. With EC, following the practice adopted in Midway, a processor needs to acquire read-only locks for the data it needs to read. Extra read-only locks occur in five out of our six applications: SOR, Water, Barnes-Hut, IS, and 3D-FFT. SOR acquires read-only locks on the boundary rows of the matrix. Water acquires (per-molecule) read-only locks on the displacements during the force computation and (per-molecule) read-only locks on the forces during the displacement computation. In Barnes-Hut, read-only locks are acquired on the cell and body structures in the load balancing and force computation phases. IS acquires a read-only lock on the shared array of buckets during the second phase of the program where each processor computes the global ranking of its keys. 3D-FFT acquires read-only locks on the parts of the matrix to be read as input to the transpose separating the second and the third one-dimensional FFT. This data is not contiguous in memory, and therefore requires support for binding non-contiguous pieces of memory to a single lock for efficiency.

A potential alternative to the use of read-only locks is to associate the data to be read in a phase with the barrier at the beginning of the phase. However, in SOR, Water, 3D-FFT, and Barnes-Hut, each processor reads a relatively small and often distinct part of the data set in a given phase (e.g., in the displacement computation phase of Water, a processor reads the forces of only the molecules it owns). In such a case, the barrier would have to be associated with the union of all the data read by all the processors. With an update protocol, this union of the data has to be transferred to all the processors, resulting in unnecessary data movement. An extension to EC that allows a per-processor association of data at a barrier might address this overhead, but involves further complexity in the programming model. Further, such a modification does not apply to the force computation phase of Barnes-Hut, where at the beginning of the phase, it cannot be determined which body and cell positions will be read by a particular processor. In contrast to the above applications, in IS, all processors read the same data, and so it may be profitable to rebind the data with the barrier. We do not explore binding data locations with barriers in this paper.

The above discussion also applies to the case where in a phase, at most one processor modifies a data object. In that case, sequential consistency and LRC rely on the barrier at the end of the phase to ensure that the modification is seen by other processors. However, with EC, we use additional exclusive locks. This case occurs in SOR, the displacement computation phase of Water, the force and position computation phases of Barnes-Hut, and 3D-FFT. The alternative of associating data with a barrier and the resulting tradeoffs discussed above for the read-only locks apply to the exclusive-lock case as well.

Task queues. The use of task queues can lead to extra synchronization in EC. Consider a task queue based program where processors execute the following actions in a loop. A processor dequeues a task from the task queue and executes the task. While executing the task, the processor potentially produces data for more tasks and enqueues the newly generated tasks. With sequential consistency and LRC, the enqueue and dequeue of the tasks take place in a critical section using locks. These locks also ensure that the dequeuer of a task sees the data produced by the enqueuer of the task. Thus, no further synchronization is necessary for accessing the task data. With EC, however, the locks around the queue ensure consistency only for the queue data. The programmer must put additional acquires and releases around the writes and reads of the task data. This case occurs in Quicksort.

Lock rebinding. In some cases, it is necessary to rebind the data associated with a lock. We identify two such scenarios.

The first scenario occurs in task queue based programs where the same data locations may become part of different tasks in different parts of the program. In this case, the programmer might allocate a lock for each entry in the task queue. When a task is enqueued, the lock for the corresponding queue entry is rebound to the data associated with the new task. In contrast, sequential consistency and LRC do not require any locks for the task data. This scenario occurs in Quicksort.

The second scenario occurs when memory is re-used for different purposes. As a result, the sharing and required communication pattern in distinct parts of the program can be different. Therefore, in EC, the programmer must explicitly rebind locks with different data to reflect the new sharing pattern. In some cases, it is possible to duplicate memory instead, and thereby avoid re-use and rebinding. This scenario occurs in 3D-FFT. We chose to duplicate memory instead of paying the penalty of rebinding.

Object granularity. In EC, the granularity at which an object is bound to a lock can be a key determinant of performance. Consider an array, where each element is a complex data structure containing several fields. Possible alternatives include a lock per field per array element, or a lock per array element, or a lock for some subset of the array. The answer clearly depends on the nature of sharing and communication in the application. For example, if all fields of a data element are accessed in one phase of the program, it is preferable to have one lock for the entire structure. Then the entire structure can be accessed with a single acquire. However, if only one part of the data structure is accessed in a different phase, then associating a single lock with the entire structure transfers more data than necessary. Further, if some fields of a large subset of the array elements are accessed in a phase, it may be profitable to associate a single lock with these fields for the entire subset. A tradeoff also occurs when, in one phase of a program, part of a data structure is read by many processors while the other part is written by one processor. If only one lock is associated with this data structure, then the structure will unnecessarily bounce from one processor to another.

In our application suite, granularity was an issue in SOR, Water and Barnes-Hut. To explore the granularity issue in SOR, we use two different versions of SOR: one in which the entire array is declared shared and which we will continue to call SOR, and one in which only the boundary rows are declared shared and which we will call SOR+.

As described earlier, in the force computation phase of Water, a processor uses several per-molecule read-only locks to read the displacements of molecules. However, the displacements of molecules owned by a single processor are modified only by that processor in the previous phase. Performance may be improved by using a per-processor lock for the displacements, resulting in a single message for all the displacement reads of molecules owned by a single processor. This either requires the ability and potential overhead of associating small non-contiguous regions of memory with one lock, or it requires splitting the displacement fields from the molecule data structure into a separate array and then associating a contiguous chunk of the array with a lock. We experimented with this approach and report the results in Section 7.2. Note that a similar approach cannot be applied to the reads of the forces in the displacement computation phase of Water, because in the preceding force computation phase the forces are modified by different processors whose identity cannot be determined statically. Thus, a per-molecule lock for the forces is still required.

In Barnes-Hut, the force and position fields of the body data structures are accessed differently in different phases of the program. However, in this case, the choice of granularity was guided by correctness requirements. In one phase, different fields of two different bodies are accessed together, resulting in a nested access of locks corresponding to the two bodies. If only one lock is associated with all fields of a body, then the nested locks can result in deadlock. Therefore, the fields of a body were divided into two sets, and a lock was associated with each set. A change similar to Water involving association of locks with multiple bodies or cells is not possible with Barnes-Hut because at the beginning of a phase it is not known which parts of the data structures that require read-only locks will be accessed.

These granularity issues do not arise in LRC because it does not need the locks described above.

4 Write Trapping

Both EC and LRC need mechanisms for detecting what shared data is changed during a particular execution interval, for it is the modified data that is communicated at synchronization points. We used two basic mechanisms for write trapping, *compiler instrumentation* and *twinning*. In this section, we explain these mechanisms and how they are adapted for EC and LRC.

4.1 Compiler Instrumentation

Compiler instrumentation of the code requires the compiler to emit extra code to set a dirty bit on writes to shared memory. A dirty bit that is set indicates that the corresponding shared data has been modified. Although always referred to as a dirty *bit*, each dirty bit actually takes up a word of memory, for reasons explained in Section 5.

As the overhead to set the dirty bit is incurred on every shared write, we have to carry out this operation as fast as possible. The approach we follow is identical to the one presented by Zekauskas et al. [13]. Shared data is allocated from large, fixed size *regions*. A region is made up of three parts. At the head is a code template, which consists of a set of routines that set the dirty bits for stores of different granularity (word, double-word, etc.). The actual shared data space is next, followed by space for the dirty bits corresponding to the data. On a shared write, the dirty bit code inserted by the compiler vectors to the appropriate template code depending on the store granularity. The template code sets the dirty bit corresponding to the location being modified.

We made modifications to the front and back ends of the Gnu C compiler (gcc) to make it emit extra code after a shared write. The front end modification consisted of adding a new type qualifier *shared*. This keyword indicates that the data it qualifies falls in the shared address space of the process. No restrictions are placed on the use of this keyword (pointers and complex types using *shared* are permitted).

In the back end, we scan the RTL (register transfer language) description of each function in the source program. After a shared write, we insert code to vector to the appropriate template code. The store granularity determines the code that is inserted. The inserted code computes the beginning of the template from the store address, and branches to the code within it.

For EC, the extra code inserted after a shared write by our compiler, and the template code itself, are identical to the Midway codes [13].

Differences between EC and LRC. When shared writes are instrumented, write collection requires scanning the dirty bits to determine which ones are set. In EC, when a lock is acquired, we only need to scan the dirty bits of the shared data object associated with the lock. As there is no such association in LRC, we would need to scan the entire shared data region, although only a small portion of it may have been updated. To avoid this problem, we use a hierarchical dirty bit scheme for LRC. This scheme sets a dirty bit for the page that is modified, in addition to setting the dirty bits for the words being modified.

Thus, for write collection, we need to scan the word-level dirty bits only for those pages for which the page-level dirty bit is set. A similar strategy could be useful for EC if the data structure associated with a lock is large and is sparsely updated between synchronization operations. Our application suite does not include such a case, and therefore we did not implement a hierarchical dirty bit scheme for EC.

Optimization of Instrumentation Code. Naive instrumentation of every shared write as suggested above, leads to suboptimal code for a number of programs. For instance, when consecutive elements of an array are updated inside a loop, the corresponding compiler inserted software dirty bit writes will also appear inside this loop.

By breaking this loop into two separate loops, one for setting the software dirty bits, and one for updating the shared data, the per-unit cost for setting the dirty bits can be reduced. In addition, the cache behavior is also improved, leading to a more efficient execution. Although current compilers are able to perform this kind of transformation, our compiler cannot, so we hand-modified the code to examine its effects.

4.2 Twinning

Twinning makes a copy of the shared data for the system, called the "twin", and later compares the user copy of the shared data to the twin to discover which elements were modified.

Twinning for EC. Our implementation of EC using twinning distinguishes between small and large objects, with the boundary between the two drawn at the size of a virtual memory page. For small objects, we make a copy of the object as soon as a write lock is requested on the object. For large objects, we make a virtual copy using copy-on-write techniques as follows. When the write lock is acquired, we write-protect the pages corresponding to the object using the virtual memory hardware. When the page is written, we make a physical copy (the twin), and unprotect the page in user space.

Our implementation differs from the Midway VM implementation of EC [13] in that they do not distinguish between large and small objects, thereby taking a protection fault on each first write to a shared object. We avoid taking this fault for small objects, based on the assumption that when a write lock is acquired, the object associated with the lock is likely to be written. A potential disadvantage of our approach is that if the object is not written, we perform an unnecessary twin (and a diff later). However, since the object is small, this overhead is small as well.

Twinning for LRC. Our twinning implementation of LRC uses virtual memory protection as used for large objects under EC. The small object approach does not apply here because there is no notion of data objects being associated with synchronization objects.

4.3 Discussion

The performance differences between compiler instrumentation and twinning depend on the number of writes to shared memory, their granularity, and their distribution in the shared address space. With a large number of writes, compiler instrumentation becomes expensive. If, however, compiler instrumentation can be done at granularities larger than a word, then it has an advantage because with twinning the comparisons are always at a word granularity. Furthermore, if the writes to shared memory are sparse, the twinning approach must make copies and comparisons over sizable areas of memory, and many protection faults may occur.

Finally, the decision of compiler instrumentation vs. twinning for write trapping also affects write collection, specifically for LRC. With twinning in LRC, write collection need scan only the twinned pages. With compiler instrumentation in LRC, the hierarchical nature of the scheme alleviates some overhead; nevertheless, write collection must still check at least one dirty bit per page in the data set.

5 Write Collection

When a synchronization request arrives, the requester must be informed of changes made to shared data. Write collection involves determining what data needs to be sent. We consider two methods, *timestamping* and *diffing*.

5.1 Timestamping

In both EC and LRC, each *block* of one or more consecutive words is assigned a timestamp, which is used to determine what data needs to be exchanged. A block is the resolution at which write trapping is done. For compiler instrumentation a block is either a word or a double-word. For twinning a block is always a single word. The notion of a timestamp is different for EC and LRC, because of differences in the two models.

Timestamping for EC. We use the notion of an *incarnation number* associated with each lock, as introduced in Midway [3]. Every time a lock is transferred, its incarnation number is incremented.

A timestamp is associated with each block in the shared address space (i.e., each word or double-word). When it is discovered that a block has changed (using the write trapping methods explained in Section 4), we set that block's timestamp to the current value of the incarnation number for the lock with which the block is associated.

With the compiler instrumentation approach, the data area for the software dirty bits and the data area for the timestamps are the same. Storage for this timestamp is the reason that each dirty "bit" takes up a word of memory, as mentioned in Section 4.

When acquiring a lock, the requesting processor includes its incarnation number for the lock in the acquire message. The responding processor sends back its incarnation number for the lock, all of its blocks of shared data associated with that lock that have a larger timestamp value than the lock incarnation number in the request, and the timestamps corresponding to the blocks. For the timestamps, only one value is sent for each run (i.e., each sequence of blocks with the same timestamp). The acquiring processor sets its incarnation number for the lock to the one received in the response message plus one, and updates its memory and timestamps with the data blocks and the timestamps received in the message.

Timestamping for LRC. The LRC model requires a more complicated timestamping procedure. The approach used in EC, timestamping updates to shared data with lock incarnation numbers, does not work for LRC, because there is no notion of data being associated with a lock. Instead, LRC uses *interval indices* as follows. The execution of each process is divided into *intervals*, each denoted by an *interval index*. Every time a process executes a release or an acquire, a new interval begins and the interval index is incremented. Similar to EC, in LRC a timestamp is associated with each shared block (i.e., each word). Unlike EC however, the timestamp for a block is a pair (p, i) consisting of a processor identifier and an interval index for that processor. These values indicate that processor p wrote the current value of the block during its interval i. For each block for which a change is detected at the end of an interval, the corresponding entry in the timestamp array is set to the tuple *(local processor id, current interval index)*.

Intervals of different processes are partially ordered [1]: (i) intervals on a single processor are totally ordered by program order, and (ii) an interval on processor p precedes an interval on processor q if the interval of q begins with the acquire corresponding to the release that concluded the interval of p. This partial order can be represented concisely by assigning a *vector* to each interval. [1] This vector contains one entry for each processor. In the vector for interval i of processor p, the entry for processor p is equal to i. The entry for processor $q \neq p$ denotes the most recent interval of processor q that precedes the current interval of processor p according to the partial order.

When processor p acquires a lock from processor q, p sends its current vector to q. As part of the lock grant message, q returns its current vector. Processor p computes its new vector as the pairwise maximum of its previous vector and the vector returned by q. Processor q further piggybacks on the lock grant message to p, *write notices* for all intervals named in q's current vector but not in the vector received from p. A write notice is an indication that a page has been modified in a particular interval, but (with an invalidate protocol) it does *not* contain the actual modifications. The arrival of a write notice for a page causes the processor to invalidate its copy of that page. A subsequent access to that page causes an access miss, at which time the modifications are propagated to the local copy.

When taking an access miss, the faulting processor includes its vector in the page fault message. The processor responding to the request then sends the timestamps and the data for the blocks in the page which have a timestamp (p, i) such that i is larger than the interval index for p in the vector received in the page fault message. Like in EC, only one timestamp value is sent for each run (i.e., each sequence of blocks with the same timestamp). The faulting processor uses the received information to update its memory for this page and its timestamps for the page's blocks.

5.2 Diffing

With diffing, a record called a *diff*, is created. A diff is a runlength encoding of the actual changes to the object (in EC) or to the page (in LRC).

Diffing in EC. As in timestamping for EC, each lock has an incarnation number, which is incremented every time the lock is transferred. At that time, a diff is made encapsulating the changes to the associated shared data object, and this diff is tagged with the incarnation number.

When a process requests a lock, it sends its incarnation number for the lock with the lock request message. When the current owner grants the lock, it sends with the lock grant message all the diffs that have an incarnation number larger than the one in the lock request message, and deletes these diffs. The requester applies the diffs in incarnation number order to its memory, and also saves them for possible future transmission to other processors.

Diffing in LRC. As in timestamping, LRC maintains interval indices and a vector of interval indices, and sends write notices during synchronization. For each page for which a change has been detected during an interval, a diff is made. At the time of an access miss, the faulting processor requests those diffs for which it has received write notices with a vector dominating the current vector for its copy of the page. The requester applies the diffs in vector order to the page, and also saves them for possible future transmission to other processors.

5.3 Discussion

Both EC and LRC use the *lazy diffing* optimization, making diffs only when the data is requested rather than at the end of an interval.

Diffing is used in combination with twinning only, and not in combination with compiler instrumentation, because the latter combination would incur the

[1] In earlier papers (e.g.,[8]), the term *vector timestamp* was used to denote this vector, but we do not use this term here to avoid confusion with the use of the word *timestamp* to denote the combination of processor identifier and interval index.

memory overhead of both the software dirty bits and the diffs.

The potential performance differences between timestamping and diffing can be understood by separately considering the computation and communication overhead of the two approaches.

The timestamp approach incurs more computation overhead because each time a request for data comes in, a scan of the timestamps is required. With diffing, the diff is computed only once, and is returned immediately in response to subsequent requests for it. If the data is requested $n-1$ times (by each of $n-1$ other processors), the timestamp method requires $n-1$ scans of the timestamps while the diffing scheme requires only one diff creation.

The timestamp approach incurs a lower communication overhead than diffing for migratory data, but a higher communication overhead than diffing for data that is shared at a fine granularity.

For migratory data, that passes in a round-robin fashion from one processor to the next and gets updated by every processor, diffing sends more data than timestamping. In the diffing approach, a diff will be created at each processor. After the initial startup, with n processors, $n-1$ diffs are passed on to the next processor every time, even though some or all of those diffs might contain changes to the same data. In contrast, with the timestamping approach, only a single value is sent for each data item changed.

On the other hand, consider programs with fine-grain sharing where different elements of the transferred data (a page with LRC and a data object associated with synchronization in EC) are updated by different processors. In this case, the number of data elements in a run with the same timestamp will be small and so several timestamps need to be communicated. Moreover, for LRC, each of the timestamps consists of a processor identifier and an interval index. Diffing requires only a single vector of interval indices for the entire diff.

6 Experimental Environment

Our experimental environment consists of 8 DECstation-5000/240's running Ultrix V4.3. Each machine has a Fore ATM interface that is connected to a Fore ATM switch. The connection between the interface boards and the switch operates at 100-Mbps; the switch has an aggregate throughput of 1.2-Gbps. The interface board does programmed I/O into transmit and receive FIFOs, and requires fragmentation and reassembly of ATM cells by software. Interrupts are raised at the end of a message or a (nearly) full receive FIFO. Unless otherwise noted, the performance numbers describe 8-processor executions on the ATM LAN using the low-level adaptation layer protocol AAL3/4.

In order to make a fair comparison, the various implementations share as much code as possible. In particular, the location and synchronization aspects of locks and barriers are implemented in the same way, although the consistency aspects differ. Furthermore, all communication and memory management aspects also share the same code.

Each lock has a statically assigned manager processor. Assignment of locks to processors is done in a round-robin way to distribute the load. The identity of a lock's manager can be derived from its lock identifier. A request for a lock is first sent to the manager, and then forwarded to the processor that last requested the lock. If the lock is free, it is granted by a message from the last owner to the requester. If not, the request is queued and granted when the lock becomes available.

Barriers also have a statically assigned manager. When a processor arrives at a barrier, it sends a message to the barrier manager. When the barrier manager has received arrival messages from all other processors and has itself arrived at the barrier, it lowers the barrier by sending a departure message to all other processors. Upon receipt of that departure message, a processor continues its execution after the barrier.

All implementations rely on Unix and its standard libraries to accomplish remote process creation, interprocessor communication, and memory management. Interprocessor communication is done through the socket interface using the AAL3/4 protocol. AAL3/4 is a connection-oriented, unreliable message protocol specified by the ATM standard. We use operation-specific, user-level protocols on top of AAL3/4 to insure delivery. To minimize latency in handling incoming asynchronous requests, we use a `SIGIO` signal handler. After the handler receives the message, it performs the request and returns. To implement memory protection, we use the `mprotect` system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a `SIGSEGV` signal.

Table 2 presents the parameters used for the applications used in the experiments. In order to provide a fair comparison between compiler instrumentation and diffing for SOR and SOR+, we initialized all internal elements to nonzero values in such a way that they change on every iteration.

7 Performance of EC vs. LRC

In Section 7.1 we qualitatively analyze the factors that may lead to different performance for EC and LRC. In Section 7.2 we turn our attention to the applications. For each application, we compare the best implementation of EC and LRC, and explain the differences, based on the factors outlined in Section 7.1.

Application	Data Set Size
SOR	1000x1000 floats
SOR+	1000x1000 floats
QS	262,144 integers, cutoff 1024
Water	343 molecules, 5 iterations
Barnes-Hut	8,192 bodies, 5 iterations
IS	$N = 2^{20}$, $B_{max} = 2^9$, 10 rankings
3D-FFT	64x64x32

Table 2 Application Parameters

7.1 Expected Performance Differences

We identify five factors that can lead to a performance difference between EC and LRC.

Extra synchronization. EC can result in lower performance than LRC because EC requires more synchronization as discussed in Section 3.3. The extra synchronization translates into more messages for EC.

Update vs. invalidate. The update protocol used in EC reduces the number of access misses and the number of message exchanges, compared to the invalidate protocol used in LRC.

Prefetching. LRC makes a whole page consistent on an access miss. In EC, only the data that is associated with a lock gets updated at a lock acquire. If multiple data items associated with different locks lie within the same page, then EC requires a message exchange for each data item, while LRC makes the entire page consistent on the first access miss to the page. To achieve this prefetch effect, LRC does not need to transmit the entire page. Only the modified elements are transmitted.

False sharing. The primary potential performance advantage for EC arises because at a lock acquire, only data associated with the lock needs to be made consistent. In contrast, an LRC system needs to ensure consistency for all data objects. This can result in less data being transferred in EC than in LRC. Consider, for example, the following scenario. Two processors access two different parts of the same page, and each of these parts is bound to a different lock. The processors write their respective parts in one phase and read the same parts in the next phase, with a barrier separating the two phases. No communication for the page is needed for the second phase, since each processor has the most recent value of the data that it will read. With EC, no communication takes place. With LRC, at the barrier before the second phase, the page must be invalidated at both processors. Our implementations of LRC are, however, not subject to the "ping-pong" effect, because they allow multiple concurrent writers per page [4].

Rebinding. The rebinding effect is an artifact of the extra synchronization required in EC (see Section 3.3). In the EC implementations, the acquire message carries the acquirer's value of the incarnation number for the lock. This value indicates to the releasing processor the last time that the acquiring processor saw the values of the data bound to the lock. The releasing processor then forwards only the data (bound to the lock) that has changed since the acquiring processor last held the lock. After a lock has been rebound to a new set of memory locations, neither the acquiring nor the releasing processor knows which part of the new data bound to the lock is consistent at the acquiring processor. Therefore, the releasing processor has to conservatively send all data that is rebound to the lock, potentially resulting in unnecessary data transfer. This issue does not arise in LRC since there is no notion of rebinding.

7.2 Application Performance

Table 3 compares EC vs. LRC. For each application, the columns `EC` and `LRC` show the execution times on 8 processors for the best implementations of EC and LRC. Columns `EC Imp.` and `LRC Imp.` show the implementation that achieved the best performance. EC-ci and LRC-ci denote the compiler-instrumented version of EC and LRC, respectively. LRC-diff, EC-diff, LRC-time, and EC-time perform twinning for write trapping. The "diff" and "time" descriptions represent whether the implementation uses diffs or timestamps for write collection. Column `1 proc.` shows the single-processor execution time of the sequential version of the application.

Two applications (IS and SOR) show very little difference between EC and LRC. For two applications (Water and Barnes-Hut), LRC is better than EC. For two applications (QS and 3D-FFT), EC is better than LRC.

The execution time of **SOR** is approximately the same on the best LRC implementation (LRC-diff) as on the best EC implementation (EC-time). Compared to EC, the positive effects of prefetching balance the negative effects of false sharing in LRC. False sharing occurs because the size of a row is a little short of the size of a page. Therefore, EC-time transfers less data (5.7MB) than LRC-diff (5.8MB). Prefetching occurs in LRC because a row of the matrix is laid out with all its red elements first and its black elements next. The data size is such that the red and black elements can both be on the same page. In LRC-diff, when a processor fetches the red part of the row from its neighbor below, this neighbor processor has often already finished computing the values of the black part of that row for the current phase. This new value for the black part is fetched along with the red values. Therefore, in the next phase, there is no miss for the black row. EC-time, on the other hand, needs to communicate to get a read-only lock each time it accesses a boundary black or red row. This is borne out by the difference in the number of messages (6936 for LRC-diff vs. 10498 for EC-time).

Making only the boundary elements shared, as in SOR+, does not affect the performance of EC-time by much, since twinning does not take place unless the data is actively shared.

For **QS** the best EC execution time (EC-diff) is 14% lower than the best LRC execution time (LRC-time).

App.	1 proc.	EC	LRC	EC Imp.	LRC Imp.
SOR	86.10	13.23	13.14	time	diff
SOR+	86.10	13.22	n/a	time	n/a
QS	47.89	8.33	9.66	diff	time
Water	61.21	18.25	12.41	ci	diff
Barnes	133.76	63.07	37.75	time	diff
IS	10.27	1.81	1.86	time	time
3D-FFT	39.82	8.32	9.23	ci	diff

Table 3 EC vs. LRC: Execution Times (in Seconds) for Best Implementation of Each Model

The task size in QS is not a multiple of the page size, resulting in false sharing for LRC. This observation is borne out by comparing the amount of data transferred in EC-diff (3.4MB) vs. LRC-time (7.1MB). QS requires rebinding of locks in EC, but any potential performance effect is completely masked because all of the data associated with the lock has been modified by the processor inserting the task in the queue. Hence all the data needs to be moved anyway.

Water's execution time is about 33% better on the best LRC implementation (LRC-diff) than on the best EC implementation (EC-ci). The dominant effect in Water is prefetching in the phase where a processor computes the new displacements of its own set of molecules. EC-ci requires a write lock for every molecule in order to update the displacements. LRC-diff, on the other hand, does not require a lock in this phase, because the programmer knows that the displacements are modified by a single processor. In LRC-diff, whenever there is a miss, the entire page is updated, including the records for all molecules on that page, resulting in fewer messages than EC-ci. There is a similar prefetching effect in the phase where the forces are computed from the displacements. Fewer lock accesses and prefetching results in a much reduced message count on LRC-diff (11381) compared to EC-ci (69422).

Both the EC version and the LRC version of Water could be further optimized by reorganizing the data structures. In the commonly used version of Water the molecules are represented by an array of records, each of which contains the displacements and the forces for a single molecule. By reorganizing this single array into two arrays, one which contains the displacements for all molecules and one which contains the forces for all the molecules, better performance can be achieved. In LRC, using two separate arrays leads to better locality and less data transmitted. In EC, this restructuring allows a per-processor lock to be bound to all displacements computed by that processor, essentially achieving a prefetch effect similar to LRC for the force computation phase. This restructuring led to an execution time of 12.50 seconds for EC vs. 11.45 seconds for LRC on 8 processors. The same effect could be obtained in the original program by binding a per-processor lock to multiple regions of memory, in particular to the displacement fields in the individual records. Note that these changes do not provide the same prefetch effect for EC in the phase where the displacements are computed from the forces, because the forces are updated by different processors and therefore still require a per-molecule lock. To provide the prefetch effect in this phase, the force fields accessed by a processor would have to be rebound to a single lock.

Barnes-Hut's execution time is about 41% better on the best LRC implementation (LRC-diff) than on the best EC implementation (EC-time). This result is explained by the combination of three different factors: extra synchronization, prefetching, and false sharing. The extra synchronization and the prefetching are seen in the load balancing phase and in the force calculation phase. Both phases involve a traversal of the tree and reads of several cells and bodies. Each time a body or cell is read, EC-time requires the use of a read-only lock. LRC-diff does not need such locks. Furthermore, on every access miss for a body or a cell, LRC-diff fetches consistent copies of all the other cells or bodies in the page. It is likely that a processor that accesses a cell on a page will also access other cells on the same page. Consequently, the number of messages with LRC-diff is lower than that with EC-ci. For example, in the last iteration of the above two phases, LRC-diff sends 1851 messages while EC-time sends 3207 messages. A significant amount of false sharing occurs in LRC-diff in all phases of the program. There are several cells or bodies in one page. Typically, accessing a cell or body on a page does not imply that the processor will access all other cells or bodies on that page. As a result, EC-time transfers less data than LRC-diff (9.5MB for EC-time and 29.9MB for LRC-diff). Of the three effects described, reduced synchronization and prefetching in LRC dominate its disadvantage in terms of false sharing for this application.

For **IS** the execution time is about the same for the best EC implementation (EC-time) and for the best LRC implementation (LRC-time). The small improvement in EC occurs because of the update protocol used in EC. There is one critical section in the program that is accessed once by each processor to add its increments to the shared array. The size of the shared array is less than a page, resulting in one additional message (due to an access miss) for every processor with LRC-time.

3D-FFT's execution time is about 10% smaller on the best EC implementation (EC-ci), compared to the best LRC implementation (LRC-diff) primarily because of the update protocol used by EC. The size of a data object bound to a lock is eight pages. Furthermore, for every acquire of the object, all the eight pages get written. On EC-ci, 3D-FFT gets all the data at the acquire because of the update protocol. LRC-diff's invalidation protocol results in separate page fault requests for each page. This argument is verified through the greater number of messages seen on LRC-diff (7175) than on EC-ci (2517), while the amount of data transferred is about the same (12.9MB for LRC-diff vs. 13.0MB for EC-ci).

In summary, neither EC nor LRC perform consistently better than the other. The differences in performance seen between EC and LRC are largely due to the size of the coherence unit. In our implementations of EC, the coherence unit is the size of the data bound to a synchronization object, while for LRC, the coherence unit is the size of a virtual memory page.

For our applications, EC outperforms LRC if its coherence unit is larger than a page, as in 3D-FFT. If EC's coherence unit is smaller than a page, then EC outperforms LRC if there is false sharing, as in QS, while LRC outperforms EC if there is spatial locality resulting in a prefetch effect, as in Water or Barnes-Hut. For Water, an equivalent prefetching effect can be achieved in EC with further programming effort, as suggested in the discussion on Water above. A similar change with Barnes-Hut is more difficult, because in the force calculation phase of Barnes-Hut, the bodies and cells that will be read are not known a priori.

Most of the extra programming effort suggested for EC attempts to bring in exactly the right data to a processor. Similar programmer or compiler effort may also have a beneficial effect for a more sophisticated implementation of LRC. There it could be used to selectively update certain pages rather than using a simple invalidate protocol.

8 Performance of Write Trapping and Write Collection

Tables 4 and 5 contain the performance results for the various combinations of write trapping and write collection examined for EC and LRC, respectively. [2]

8.1 Write Trapping

In order to evaluate write trapping performance, we compare the implementations with identical write collection mechanisms, i.e., EC-ci and EC-time for EC, and LRC-ci and LRC-time for LRC, all of which use timestamping for write collection.

For EC (see Table 4), EC-ci performs better only when the dirty bit represents a data item larger than 4 bytes, the granularity of comparison for the twinning version. The programs for which this is the case are 3D-FFT and Water, both of which use an 8-byte granularity. With an 8-byte granularity, the number of dirty bits that need to be scanned in the write collection phase is halved, reducing computation. Our current implementation of timestamps using twinning uses a granularity of 4 bytes even when the object granularity is higher.

For LRC, the trade-offs between the compiler-instrumented and twinning versions are more well-defined (see Table 5). The compiler-instrumented version performs worse than the twinning versions for all the programs. We attribute the inferior performance of compiler instrumentation for LRC to a number of factors including the need to scan over larger ranges of timestamps for write collection, and the increased cost of setting dirty bits because of their hierarchical nature.

	EC-ci	EC-time	EC-diff
Trapping	Comp. Ins.	Twinning	
Collection	Timestamps	Timestamps	Diffs
SOR	14.86	13.23	13.28
SOR+	14.09	13.22	13.25
QS	9.71	8.50	8.33
Water	18.25	19.21	19.73
Barnes-Hut	63.15	63.07	64.89
IS	1.86	1.81	2.01
3D-FFT	8.32	9.59	8.68

Table 4 Execution Times (in Seconds) for Various Combinations of Write Trapping and Write Collection in EC

	LRC-ci	LRC-time	LRC-diff
Trapping	Comp. Ins.	Twinning	
Collection	Timestamps	Timestamps	Diffs
SOR	18.87	13.41	13.14
SOR+	n/a	n/a	n/a
QS	26.44	9.66	10.04
Water	17.11	13.05	12.41
Barnes-Hut		57.59	37.75
IS	2.42	1.86	2.06
3D-FFT	13.95	10.32	9.23

Table 5 Execution Times (in Seconds) for Various Combinations of Write Trapping and Write Collection in LRC

All results for EC-ci and LRC-ci in this paper are with the compiler optimization discussed in Section 4.1 in place. Except for SOR, where the compiler optimization achieved a 16% execution time improvement, the benefits were minor (5% for SOR+, 2% for Water, and no improvement for the other programs).

8.2 Write Collection

In order to evaluate write collection performance, we compare the implementations with identical write trapping mechanisms, i.e., EC-time and EC-diff for EC, and LRC-time and LRC-diff for LRC, all of which use twinning.

For EC, the timestamping version performs the best for programs with migratory data that send multiple overlapping diffs, such as IS and Water. For example, the diffing version of IS sends 4 times as much data as the timestamping version (1.3MB for EC-diff vs. 0.3MB for EC-time) at 8 processors. Hence, the execution time of the timestamping version is 10% better. For programs that require only a single diff to be communicated at any acquire (for example, with producer-consumer data), such as QS and 3D-FFT, the diffing version performs the best because it requires less computation, while the amount of data sent stays the same. For SOR, SOR+, and Barnes-Hut, there is little difference between timestamping and diffing.

For LRC, SOR and SOR+ continue to show little difference, IS again shows better performance with timestamping, as does 3D-FFT with diffing. For Barnes-Hut, however, diffing performs much better than timestamping. Also, Water's performance with diffing has become better than with timestamping. We attribute this result to the need to communicate a large number of timestamps for these applications which exhibit relatively fine-grain sharing within a page (see Section 5.3).

[2] The execution time for Barnes-Hut LRC-ci was not available at the time of writing.

9 Related Work

The paper by Zekauskas et al. [13] studies the difference between a compiler instrumentation based and a virtual memory based implementation of EC. There are a number of differences between their study and ours. First, we consider both EC and LRC, and offer a comparison between the two. Second, our EC twinning implementation results in far fewer protection faults than their virtual memory based implementation, because for small objects we copy the object immediately when a write lock is acquired, rather than write-protecting it. Our implementation performs worse only if the object is not written, a rare occurrence. Even in that case the cost is small because the object is small and therefore copying and diffing it is not very expensive. Third, they investigate only the combinations of write instrumentation and timestamps, and the combination of twinning and diffing. We investigate also the combination of twinning and timestamps, which is seen to be advantageous for applications dominated by migratory data. The execution times in this paper should not be compared directly to their results, because of differences in processor speed (40Mhz vs. 25Mhz) and differences in communication speed and page fault overhead in the underlying operating systems (Ultrix vs. Mach 3.0).

Castro et al. [5] compare DiSOM, an object-oriented parallel programming system using EC, to TreadMarks, a DSM system using LRC. They conclude that DiSOM provides better performance than TreadMarks for a number of applications (matrix multiply, SOR, TSP, and Water). Superficially, these results appear to contradict the results in this paper. However, the nature of the comparison in Castro et al. is very different from the comparison made in this paper. In addition to using EC, DiSOM relies on a number of other techniques to achieve good performance. These techniques include taking advantage of the object-oriented nature of the system to avoid write trapping, and fine-grain control over communication, such as per-object `pack` and `unpack` routines.

10 Conclusions

For the environment and the applications examined, neither EC nor LRC consistently outperforms the other. The differences are largely due to the different coherence units in these software DSM systems. EC outperforms LRC if the effect of false sharing within a page dominates, or if the coherence unit is larger than a page. LRC outperforms EC if the page-sized coherency unit results in prefetching due to locality of future accesses. Given that there is no clear winner in terms of performance, the choice between the two models should probably be guided by programmability considerations.

In terms of programmability, we found that, in general, writing a program to run correctly on EC required more effort than on LRC. For the programmer who is writing a parallel program from scratch, this effort may not be significant since most of the sharing behavior is well-understood. However, for the programmer who needs to port a program written for sequential consistency to EC, the effort required can be substantial since detailed data access behavior needs to be understood. The effort in writing a program for good performance with EC, however, seems to be considerable for many programs for both the initial programmer and the programmer concerned with porting.

We also assessed the performance of two write trapping and two write collection techniques. For write trapping, compiler instrumentation pays off in EC only when the granularity of sharing is greater than a word, since the number of dirty bits scanned, and therefore the computation, is reduced. In the case of LRC, compiler instrumentation has worse performance than twinning for all the applications that we examined. For write collection, we have demonstrated that for EC, timestamps perform better for migratory data due to reduced communication requirements. Diffing sends more data because of multiple overlapping diffs being communicated. For other types of data, the higher computation overhead and communication overhead due to sending multiple timestamps result in poorer performance for timestamping. These overheads for timestamping are more important in LRC, and sometimes dominate the gains of timestamping for migratory data.

References

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE TPDS*, June 1993.

[2] D. Bailey et al. The NAS parallel benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

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

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM TOCS*, August 1995.

[5] M. Castro et al. Efficient and flexible object sharing. Technical report, INESC, July 1995.

[6] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE TSE*, June 1990.

[7] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th ISCA*, May 1990.

[8] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th ISCA*, May 1992.

[9] P. Keleher et al. An evaluation of software-based release consistent protocols. *JPDC*, October 1995.

[10] P. Keleher et al. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter Usenix Conference*, January 1994.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, November 1989.

[12] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[13] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the 1st OSDI Symposium*, November 1994.