Scope Consistency : A Bridge between Release Consistency and Entry Consistency

Liviu Iftode, Jaswinder Pal Singh and Kai Li
Department of Computer Science, Princeton University, Princeton, NJ 08544

Abstract

Systems that maintain coherence at large granularity such as shared virtual memory systems, suffer from false sharing and extra communication. Relaxed memory consistency models have been used to alleviate these problems, but at a cost in programming complexity. Release Consistency (RC) and Lazy Release Consistency (LRC) are accepted to offer a reasonable tradeoff between performance and programming complexity. Entry Consistency (EC) offers a more relaxed consistency model, but it requires explicit association of shared data objects with synchronization variables. The programming burden of providing such associations can be substantial.

This paper proposes a new consistency model for such systems, called *Scope Consistency* (ScC), which offers most of the performance advantages of the EC model without requiring explicit bindings between data and synchronization variables. Instead, ScC dynamically detects the associations implied by the programmer, using a programming interface similar to that of RC or LRC. We propose two ScC protocols: one that uses hardware support for fine-grained remote writes (automatic updates or AU) and the other, an all-software protocol. We compare the AU-based ScC protocol with Automatic Update Release Consistency (AURC), a modified LRC protocol that takes also advantage of automatic update support. AURC already improves performance substantially over an all-software LRC protocol. For three of the five applications we used, ScC further improves the speedups achieved by AURC by about 10%.

1 Introduction

Systems that preserve coherence at a large granularity, such as shared virtual memory systems, tend to suffer from false sharing and extra communication. Several relaxed memory consistency models [11, 8, 20, 3] have been proposed to alleviate the performance in shared virtual memory systems. However, models with increasing effectiveness also increase programming complexity. The challenge is to define models that maximize the performance benefits of relaxed consistency while minimizing the additional programming requirements.

Protocols based on Release Consistency (RC) [11] are accepted to offer a reasonable tradeoff between performance and programming complexity for shared virtual memory [7]. RC guarantees memory consistency only at synchronization points, which are marked as *acquire* or *release* operations. Modifications to shared data are globally performed [11] no later than the next release operation. Lazy Release Consistency (LRC) [20] is a relaxed implementation of RC in which coherence actions are postponed from the release to the next acquire operation. RC and LRC protocols may be implemented entirely in software or can be modified to take advantage of new network interfaces that provide hardware support for word-level *automatic updates* of remote copies of shared data upon writes [13, 5]. Automatic Update Release Consistency (AURC) [16] is one such protocol that improves performance substantially over all-software LRC, by using the automatic update mechanism provided in the SHRIMP network interface [5, 6]. The additional programming requirements imposed by RC-based protocols are small: Application programs simply need to mark all their synchronizations as acquire or release operations as appropriate.

The consistency model can be further relaxed by exploiting the association of shared data objects with synchronization variables (locks or barriers). For example, Entry consistency (EC) [3] lets the acquire operation obtain modifications only to the data that have been explicitly associated with the acquired synchronization variable — not all the shared data that have been modified by other processors — thus reducing unnecessary traffic compared to LRC or AURC. The EC model, however, requires that the coherence protocol be given the associations between data structures and synchronization variables. Thus, while EC can improve performance, the programming burden of explicitly associating synchronization with data can be substantial (since compilers cannot do this automatically today) and EC has not gained wide acceptance.

This paper proposes a new memory consistency model called **Scope Consistency** (ScC). ScC offers most of the potential performance advantages of EC, without requiring explicit association of data to synchronization variables. It introduces the concept of *consistency scope*, which effectively establishes the association dynamically and transparently. In most cases, programs that follow the programming discipline of LRC can run with ScC without modifications — their synchronization events imply the scopes. However, not all LRC programs are correct under ScC, and additional scopes may have to be defined in some cases. Nonetheless, even in these cases the extended interface for ScC is simpler to use than the EC programming interface, since scopes are associated with sections of code rather than with data.

As a consistency model, ScC is independent of implementation, and can be used with any shared virtual memory protocol. However its performance potential is realized when

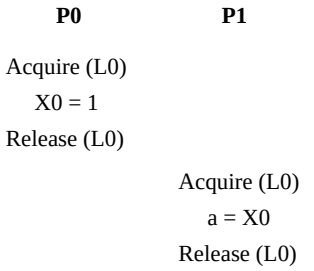


Figure 1: Programming in Release Consistency

used with a "home-based" protocol in which modifications are eagerly propagated to the home node for a page – whether supported by hardware automatic update(AU) or in software. These home-based protocols have been shown to substantially outperform earlier distributed approaches to LRC [17, 33].

To understand the performance implications we implemented an AU-based ScC and an AURC protocol within the TangoLite simulation framework [15]. We conducted detailed simulation studies with five Splash-2 [31] applications, as well as a synthetic benchmark designed to emphasize a communication pattern that illustrates the benefits of ScC. ScC reduces the number of page faults in the synthetic benchmark by 45%. For three of the five real applications, ScC improves the speedup achieved by AURC by about 10%. For the other two applications, ScC did as well as AURC. One ScC-improved application and both the unimproved ones were already ScC, while the other two applications needed program changes to comply with ScC.

2 Relaxed Consistency Models

A consistency model is essentially a contract between the memory system and the parallel program which specifies the order in which memory accesses can be executed by the system [2]. Relaxed consistency models define a set of sufficient conditions or rules such that if the program obeys these rules, the system guarantees sequentially consistent memory behavior. The goal of consistency relaxation is to provide the system a certain degree of freedom in ordering memory operations, thus reducing the impact of data access latency and improving the overall performance. This is particularly important in systems that preserve coherence in software and at a large granularity. Unfortunately, relaxed consistency models increase programming complexity, leading to a tradeoff with performance. In the following subsections we revisit release consistency [11] and entry consistency [3], two important consistency models which are supported in software shared memory systems.

2.1 Release Consistency

The *Release Consistency(RC)* model was initially introduced for hardware cache-coherent multiprocessor [11]. It has since become the cornerstone of software shared virtual memory systems [8, 20, 19, 17, 21].

Release consistency distinguishes between ordinary memory accesses and synchronization accesses defined as either *acquire* or *release*. To make this possible the program must be *properly labeled* [11] (Figure 1). Only the synchronization accesses are strictly ordered (either sequentially [22] or processor consistent [14]). Ordinary memory accesses to different locations are ordered only with respect to synchronization accesses and can be completely reordered between them. The following two definitions are reformulated after [9, 27, 11]. A complete set of definitions for terms used in consistency models can be found in [2].

- A write is *performed with respect to a process P* when a read issued to the same address by P will return the value stored by that write (or a subsequent write to the same location).
- A read is *performed with respect to a process P* when the issuing of a write to the same address by P cannot affect the value returned by the read

The consistency conditions for release consistency [11] can be formulated as following:

1. Before an ordinary memory access is allowed to perform with respect to any other process, all previous acquire operations must have completed successfully.
2. Before a release operation is allowed to perform with respect to any other process, all previous ordinary accesses must have completed.
3. Acquire and Release operations must be processor consistent with respect to one another.

The combined effect of these rules is that all memory accesses preceding the release of a lock (say) are guaranteed to be performed before all memory accesses which follow the acquire of the lock. This is the only order among memory accesses which is guaranteed in release consistency.

In shared virtual memory systems release consistency can be implemented with either eager or lazy protocols. In eager release consistency, data modifications are propagated no later than the release time. In lazy release consistency, on the other hand, a timestamp-based protocol is used to propagate the update information lazily at the acquire time following the causal order given by the chain of release-acquire operations. By not propagating modification information globally at a release, but postponing it until the next acquire, LRC further reduces the extra communication due to false sharing.

2.2 Entry Consistency

The *Entry Consistency(EC)* model was proposed for the Midway software shared memory system [3]. Midway is a variable- or region-based rather than a page-based system. Entry consistency requires each ordinary shared variable to be explicitly associated with some synchronization variable such as a lock or barrier which protects access to that variable (Figure 2). When a lock is acquired only the variables associated with that lock are updated. The explicit association of data to synchronization further reduces the effect of false sharing but at the same time adds substantial programming complexity compared with release consistency.

Entry consistency distinguishes between exclusive and non-exclusive accesses to a lock. To write the variables associated with a given lock, the process must own the lock i.e.

```
P0                          P1

Lock_Bind(L0, X0)           Lock_Bind(L0, X0)
Acquire_Exclusive(L0)
   X0 = 1
Release (L0)

                            Acquire_Non_Exclusive (L0)
                               a = X0
                            Release (L0)
```

Figure 2: Programming in Entry Consistency

must acquire the corresponding lock in exclusive mode. In non-exclusive mode, multiple processes can acquire the same lock but they can only read the associated variables. The consistency rules for entry consistency are the following [3]:

1. Before an acquire is allowed to perform, all updates to the guarded shared data must be performed with respect to the process.
2. When a lock is acquired in exclusive mode, no other process may acquire that lock, not even in non-exclusive mode.
3. After a lock is acquired in exclusive mode, the next non-exclusive acquire of that lock performed by any other process is allowed to perform only after it is performed with respect to the owner of that lock.

3 Scope Consistency

Scope Consistency(ScC) uses a concept called *consistency scope* to establish the relationship between data and synchronization events implicitly. The main attraction of this approach is that in most cases consistency scopes can be derived implicitly from the synchronization already present in programs that conform to release consistency, and can deliver performance improvement similar to entry consistency. When required, additional explicit scopes can be introduced through annotations, which are much easier to use than in EC because the binding is defined in terms of code sections and not variables.

3.1 Consistency Scope

A *consistency scope* is a limited view of memory with respect to which memory references are performed. That is, modifications to data performed within a scope are only guaranteed to be visible within that scope. We can think of a consistency scope as consisting of all critical sections protected by the same lock. Additionally, barriers define a global consistency scope which includes the entire program. For each consistency scope there is an *open* operation that opens the scope and a *close* operation that closes the scope. Examples are given in Sections 3.2 and 3.3. The interval during which a consistency scope is open at a given process (e.g. a given critical section protected by the lock) is called a *session*. Any modifications made within a consistency scope session become visible to processes that then enter new sessions of that scope (acquire that lock or pass a barrier). Modifications made outside the scope session are not guaranteed to be visible. Figure 3 illustrates consistency scopes. Memory is accessed through different scopes that are independent of one another. Each such perspective or scope is built out of sessions which occur on different processes. Sessions belonging to different scopes can interleave in time or overlap in the code or memory. In addition to the individual consistency scopes (whose sessions may be delineated by locks and unlocks, say), there is also a global consistency scope with regard to which all memory references are performed. The sessions of the global scope are typically delineated by barriers.

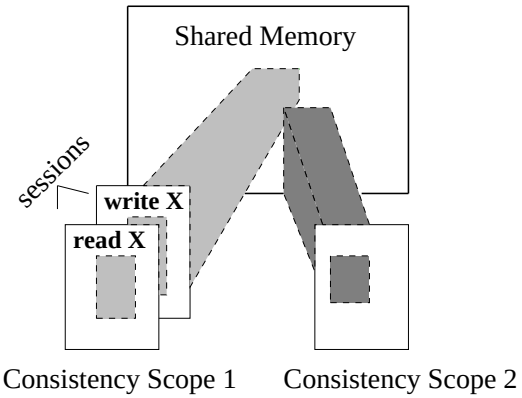


Figure 3: Consistency Scopes

To define the model we need an additional definition to distinguish a reference being *performed with respect to a consistency scope* from a reference being *performed with respect to a process*:

- A write that occurs in a consistency scope is *performed with respect to that scope* when the current session of that scope closes.

The consistency rules for Scope Consistency are:

1. Before a new session of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P.
2. A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.

The first rule establishes that after a consistency scope is opened at a process, the updates previously performed *with respect to that consistency scope* are guaranteed to be performed *with respect to that process*. In other words, all the updates which occurred in all previously closed sessions of a scope – whether performed by that process or by others – are guaranteed to be performed in the memory local to that process which opens a new session of that scope.

By forcing open operations to complete before the following memory accesses are allowed to perform, the second rule guarantees that these accesses see a view of memory that is updated according to the first rule. Let us see how consistency scopes and their sessions are indicated in programs.

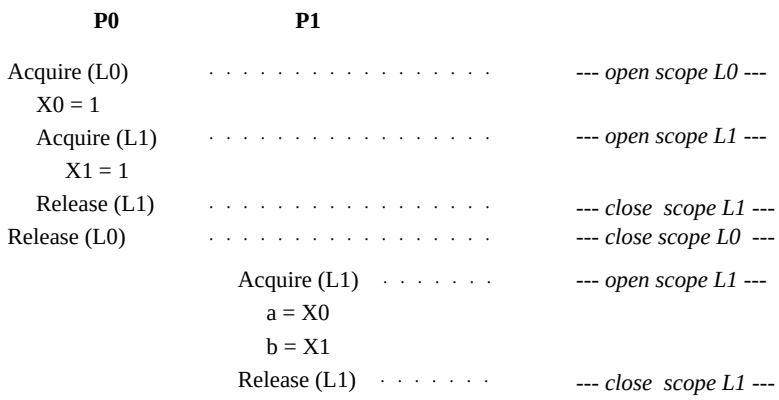


Figure 4: Programming with Lock-based Scopes

3.2 Lock-based Consistency Scopes

As mentioned earlier, in most programs ordinary synchronization events such as locks and barriers delimit consistency scopes, and no additional annotations are needed. Critical sections protected by the same lock define a consistency scope, and a particular critical section protected by that lock and entered at runtime delimits a scope session. The lock acquire opens the scope session and the lock release closes it. Barriers define a global consistency scope, initially open at all processes. Each process entering a barrier closes the global consistency scope session and opens a new one on return (barrier exit). When a process gets past a barrier, it is guaranteed to see all changes (either within or outside critical sections) made by any process before that barrier, just as in RC. Any benefits from ScC therefore occur through more specific synchronization (e.g. locks) within the region between two barriers. For flag-based synchronization an explicit *open_scope* operation can be used in conjunction with a flag set operation to delimit a consistency scope.

To reformulate the above general rules of ScC in terms of lock-based scopes we use the term *performed with respect to a lock* for *performed with respect to a lock-based scope*:

1. A write is *performed with respect to a lock or barrier* when that lock or barrier is released.

This definition implies that multiple processes can simultaneously perform writes with respect to a barrier but only one process at a time can perform writes with respect to a lock. The second definition is as before. The scope consistency rules can be formulated for locks and barriers as following:

1. Before a lock (or barrier) acquire is allowed to perform by process P, all writes performed with respect to that lock (or barrier) must be also performed with respect to P.

2. A memory access is allowed to perform with respect to a process P only after all previous acquire events at P (in program order) have succeeded.

If scope consistency semantics are attached to both locks and barriers, RC is upgraded to ScC in a very simple way: Every memory reference occurs in at least one consistency scope: the global consistency scope which is simultaneously open at all processes. In addition, a memory reference is also considered to occur in all consistency scopes corresponding to the locks which have been acquired but not yet released.

Example I

Figure 4 illustrates scope consistency with an example. When Process P0 acquires lock L0 it enters the scope defined by that lock. Process P0 writes X0 while in scope L0 and writes X1 while in scopes L0 and L1. Next, process P1 acquires lock L1, thus opening scope L1, and reads both X0 and X1. Under scope consistency, P1 is guaranteed to see P0's write to X1 assuming P1 acquires L1 after P0 releases it. However there is no guarantee that P1 will see P0's write to X0 because the scope L0 was not opened at P1 when it read X0; thus, P0 need not propagate the modification of X0 to P1 (under RC or LRC, this latter guarantee would exist since the consistency model does not distinguish among synchronization variables, and the modification would have to be propagated) . However if P1 had acquired L0 instead of L1 and if that had occurred after P0 had released L0, then both writes to X0 and X1 performed by P0 would have become visible at P1.

Not all programs that follow the programming discipline of RC are ScC-correct without modifications. We call a program ScC-correct if it observes a sequential consistency memory when run on a system that supports ScC. We believe that the following are sufficient conditions for an RC program to be ScC-correct:

1. Lock-protected modifications to shared data are not expected to be visible at a process before at least one of the protecting locks is acquired by that process.

2. Modifications to shared data that are not protected by a lock are not expected to be visible at a process before the next barrier.

It is an important area for future research to formalize these conditions and then prove they are indeed sufficient to define ScC-correct programs (as has been done for other relaxed consistency models [2, 12]).

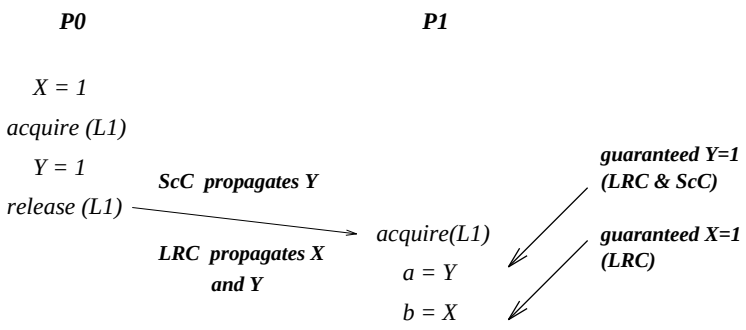


Figure 5: Scope Consistency versus Release Consistency.

Example II

Figure 5 provides an example showing the type of programming problems which can be encountered in switching from RC to ScC, using LRC as an example. P_0 writes X first and then Y; Y is written in the critical section guarded by lock L_1 , while X is written outside it. P_1 acquires L_1 after P_0 has released it.

Under LRC [20], after acquiring L_1 P_1 is guaranteed to see all writes which occurred at P_0 before the release of L_1, so the new value of both X and Y are guaranteed to be visible. On the other hand, ScC guarantees only that P_1 sees the new value of Y, because the release event ensured memory consistency over only one consistency scope, namely the one defined by all critical sections guarded by the same lock L_1. To guarantee the visibility of both X and Y, the write to X by P_0 must be performed inside the same critical section, which makes sense if the writes are semantically related. Otherwise, the write to X will be guaranteed visible at P_1 only at the next barrier.

Some programs which use task queues to distribute the computation exemplify this situation: The critical section protects the task queue modification only and not writes to data performed by that task. However, the programmer assumes that when a task is selected from the task queue, the updates by previously completed tasks (say as a result of which the current task was created) have been performed and are visible. This assumption is true under RC but not necessarily under ScC if only the task queue updates are protected by locks.

Thus, precautions may have to be taken to ensure that programs are ScC-correct. One solution, as we shall see for the Barnes-Hut application, is to make critical sections large enough to contain all modifications which are intended to be performed at the time when that lock is acquired. In some cases, this approach may not be effective since it can increase serialization and hence hurt performance. An alternative solution is to add new critical sections to protect the unprotected modifications or, better yet, to use explicit scopes in addition to the existing lock-based ones.

3.3 Explicit Scopes

Scopes can be created dynamically using explicit annotations. Figure 6 illustrates a typical example of a task-queue based program which uses explicit scopes in addition to lock-based scopes. New scopes are created to enclose the writes performed by tasks and are attached to the tasks themselves. The primitive *open_scope* with no argument creates a new scope and returns its *scope id*. Within that scope $X0$ is updated. Next, the scope is closed and its id is stored in the per-task data. When the task is scheduled at P1, the corresponding scope is explicitly opened with *open_scope* (before X0 is accessed), thus ensuring the visibility of the update performed at P0. In section 5, we will see a real application (Cholesky) which we modified using explicit scopes to make it ScC-correct.

Although the approach using *open-scope* and *close-scope* appears conceptually similar to the use of non-exclusive lock acquire in EC, there are significant differences both in terms of programming model as well as protocol implementation. Unlike in EC, in ScC explicit scopes do not require explicit binding of data to locks, thus making them much easier to use. Scopes are attached to code sections, not to data. The "bindings" to data are dynamically created based on the memory accesses issued between *open_scope* and *close_scope* while a session of the scope is open. Adding more locks is however, an alternative solution to adding explicit scopes in ScC, and its performance benefit would be very similar to the one in EC.

4 Comparison with RC and EC

Scope consistency is a relaxed consistency model situated between RC [11] and EC [3]. Let us compare it with each, using an LRC protocol to represent RC.

4.1 ScC versus RC

Both ScC and RC do not require explicit binding of data to synchronization. However, ScC assumes an implicit binding of memory accesses to scope determined from the program structure. ScC reduces to RC for applications that use only global synchronization, such as barriers, because then there is only one global scope.

Reduced False Sharing. A large coherence granularity is a source of false sharing. By postponing memory coherence from release to acquire and supporting multiple writers within the same page, LRC eliminates the effects of certain false-sharing cases compared with an (eager) RC-based shared virtual memory. But to comply with the RC model, the acquiring processor in LRC must see all the modifications to data seen so far by the releasing processor.

ScC goes further in eliminating the effects of false sharing. By supporting multiple memory scopes, ScC can postpone

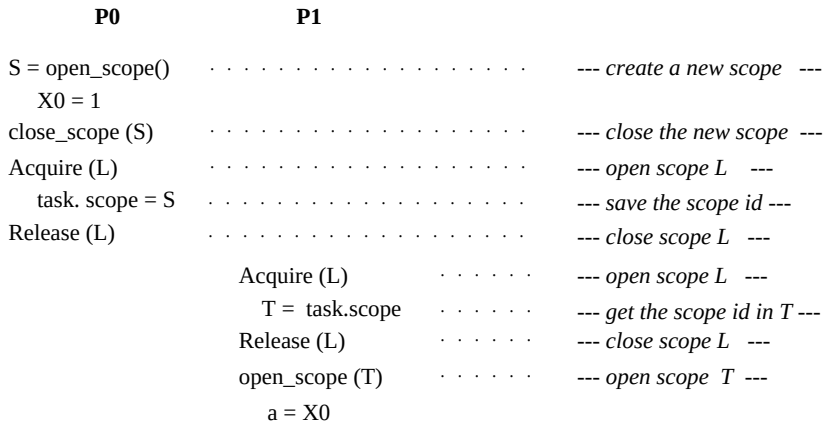


Figure 6: Programming with Explicit Scopes

seeing memory updates until the *corresponding* scope is open. This allows the acquiring processor to perform (through invalidations, say) only the modifications which occurred in the same scope.

Figure 7 illustrates the benefit of ScC over LRC in reducing the impact of false sharing. Assume that an array of N particles X, is shared by p processes. Each particle X_i is protected by a different lock L_i. Assume X_0 and X_1 lie on page *0* while X_2 and X_3 lie on page *1*.

P_0 computes an interaction between particles X_0 and X_2, then P_1 computes the interaction between X_2 and X_1. To accomplish this, P_0 modifies first X_0 and then X_2 by acquiring the corresponding locks (L_0 and L_2). When P_1 acquires lock L_2 from P_0, under the LRC model it has to see both modifications (to X_0 and X_2) since the acquire event is allowed to perform only after all previous memory accesses have been performed. Therefore, both pages (*0* and *1*) will be invalidated by P_1 in an invalidation-based protocol.

In fact, P_1 does not need to see the modification to X_0 now since it doesn't acquire L_0 and X_1 may be already up-to-date (for instance if the most recent release of lock L_1 occurred at P_1). Despite this, the invalidation that occurred on page *0* when acquiring L_2 causes a page miss when accessing X_1, due to false sharing (X_0 and X_1 reside on the same page).

In ScC, when lock L_2 is acquired by P_1 only the modifications protected by L_2 are propagated. In this case, an invalidation and the subsequent miss will be performed on page *1* but not on page *0*. The program will behave correctly since the new value for X_0 is not needed in computing the interaction between X_2 and X_1.

Programming Effort. Like RC, ScC requires explicit synchronizations which must be labeled with system-supplied primitives (e.g. lock, barrier). However, ensuring the correctness of a RC program under ScC is not trivial because it requires checking whether dependencies between data updates assumed under RC are preserved under ScC. Program modifications are sometimes necessary to make an RC-correct program ScC-correct.

4.2 ScC versus EC

Both EC and ScC relax the memory consistency model by taking advantage of the relationship between data and synchronization. Both schemes are particularly effective for applications that use a lot of point to point or mutually exclusive (lock) synchronization. However the solutions proposed by the two schemes are quite different.

Binding. In EC [3], the binding between *data objects* and *synchronization objects* is specified explicitly by the programmer. In fact, the solution proposed by EC is for variable-based software shared memory systems, close to object-based or region-based systems, not for transparent page-based systems. EC requires explicit associations of data with synchronizations because the protocol relies on this information to identify which modifications must be propagated and when. This is why EC can be implemented very effectively with an update-based protocol. In both EC and ScC binding can change during the program lifetime, but EC requires the programmer to explicitly rebind a lock to the new data it protects.

In ScC rebinding occurs automatically since it is detected by the system based on the modifications which occur in the sessions of that scope. We can say that EC binds data to locks explicitly while in ScC the binding is implicit and dynamically created by the memory accesses.

Global Synchronization. The explicit binding in EC has difficulty including global synchronization in the model [1]. The alternatives are: (1) not to bind data to barriers at all; (2) to have explicit binding as for locks; (3) to bind the entire address space. The first approach substantially limits the intuition of shared memory, since barriers do not cause modifications to become visible. It requires the programmer to employ read-only locks to cause modifications to be visible after a barrier. This is the scheme used in Midway [3]. The second approach is difficult to program because it may require multiple bindings if the barriers are used to separate different phases of the computation. The third approach requires high communication in a update-based protocol like EC uses.

ScC introduces a global scope which is a simple intuition for global synchronization. In this scope, ScC guarantees the visibility of all modifications, which is the same as the third approach discussed above. The global scope is

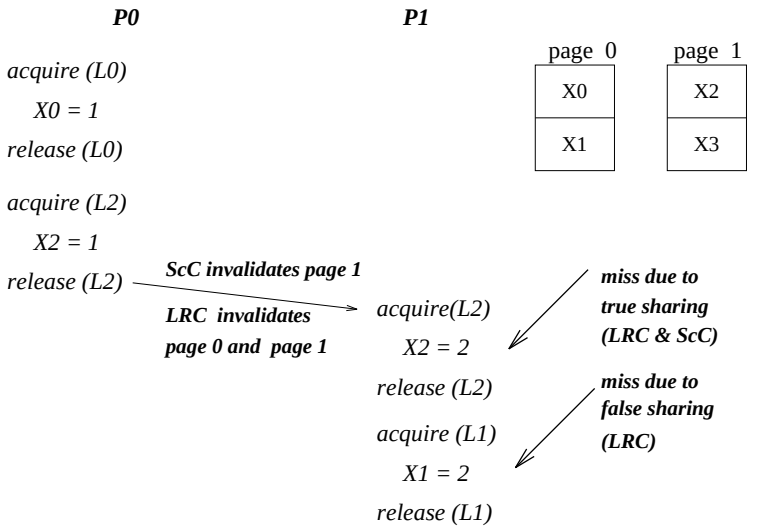


Figure 7: False Sharing in ScC versus LRC

particularly useful in ScC to avoid reasoning about update propagation in terms of transitivity of lock-based scopes. The model is simple for the programmer: after each barrier the entire shared address space is guaranteed to be coherent. In addition, unlike in EC, the global scope solution is reasonable in ScC because ScC systems will typically employ an invalidation-based rather than an update-based protocol.

Granularity. EC and ScC systems have a granularity of binding and a granularity of communication. In EC the binding granularity is of variable size because it corresponds to the *data objects* the programmer binds to *synchronization objects*. Since binding in EC must be explicitly specified, its granularity is an important factor in performance and programming complexity. Binding large contiguous pieces of memory may reintroduce the effects of false sharing and need additional lock rebindings. Small non-contiguous bindings are more efficient because they require less communication and/or less processing. But they are also more difficult to program since the programmer must establish many more bindings for each lock. Determining a good binding granularity is the task of the programmer. The communication granularity depends on how write collection is implemented. Midway [3, 32] uses software dirty bits so the amount of communication is strictly proportional to the size of the updated data.

In ScC the binding granularity is determined by the write detection granularity, which is of word size for both automatic update and all-software write detection schemes. Therefore any modification occurring in a given scope automatically binds the corresponding location to that scope. However, the communication granularity in ScC is typically of page size if ScC is implemented with a shared virtual memory protocol (see section 5) which means that in order to fetch a location which is bound to a scope, the entire page which contains that location is fetched. On the other hand, although the communication volume may be higher in ScC than in EC due to page fragmentation, the page size transfers allow ScC to benefit from potential prefetching [1].

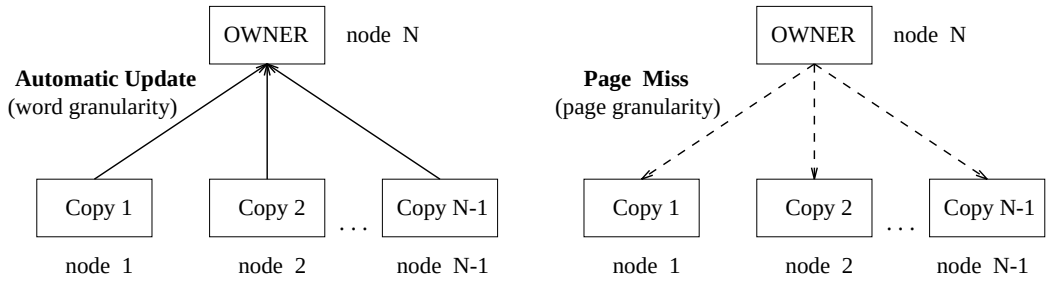
Programming Effort. ScC is potentially easier to program than EC because in most cases it doesn't require any change to RC programs. When RC programs are not ScC, adding explicit scope annotations in ScC requires similar program understanding as for adding explicit binding in EC although it is easier since it is code-oriented rather than data-oriented.

5 ScC Protocols

Scope consistency can be implemented using page-based invalidation protocols similar to those used in traditional shared virtual memory systems. For such protocols ScC reduces the impact of false sharing caused by lock-based synchronization resulting in fewer page invalidations between barriers. Savings in page invalidations produce savings in communication cost only if the number of data fetching messages and the data traffic are proportional to the number of page faults not to the number of modifications. In traditional diff-based systems like TreadMarks [19], the data traffic consists of diffs, thus it is proportional to the number of updates. A *diff* is a run-length encoding of the modifications performed by one process on a shared page. Since all the updates must be eventually performed, avoiding some page invalidations using ScC makes little difference compared with LRC because the total data traffic remains the same.

Home-based protocols are a family of protocols which can benefit from ScC because the traffic on page fault is proportional to the number of page faults, not the number of modifications to those pages. In home-based protocols, the updates performed on each local copy of a page are eagerly propagated to the home of the page. The home copy is thus kept up-to-date, so on a page fault the page is fetched from the home rather than diffs being fetched from all writers. Because each page fault requires a full-page transfer and only from the home regardless of the number of updates actually performed, the data traffic and the number of data fetch messages in home-based protocols are proportional to the number of page faults. Consequently, the impact of ScC on home-based protocols can be significant.

In what follows we introduce two ScC protocols: one which uses limited hardware support and the other an all-software protocol. They are based on similar protocols



(a) communication from the local copies to the owner's copy

(b) communication from the owner's copy to the local copies

Figure 8: Basic Communication Schemes in AURC

developed for lazy release consistency: Automatic-Update Release Consistency (AURC) [16] and Home-based Lazy Release Consistency (HLRC) [33] respectively. We begin by describing the hardware support called *automatic update*.

5.1 Automatic Update

Automatic update can be viewed as a write-through between two local memories. It allows writes to selected pages to be performed twice (doubled) both on a local memory page as well as on a remote memory page, assuming a mapping was previously established between the source and the destination pages. For example if local page A at processor 0 is mapped to page B at processor 1, then all updates which are performed locally on page A are also automatically propagated to page B at node 1. The network interface of the SHRIMP multicomputer built at Princeton [5, 6, 10], implements automatic update by mapping the selected pages on write-through caches and snooping all write traffic on the local memory bus. Writes to local memory pages which are mapped out on remote pages, are intercepted and forwarded to the remote destination by the SHRIMP network interface. The DEC Memory Channel [13] allows modifications to be explicitly propagated at word level through I/O writes.

This automatic update feature provides adequate support for a shared virtual memory implementation. AURC [16] is a protocol which implements LRC [20] with automatic update. The basic approach in AURC is to use automatic update mappings to merge updates from multiple writers on the same page into the *home*'s copy of that page (see Figure 8). That is, all copies of a page map to a fixed home, and the home copy is always kept up-to-date by automatic update. This mechanism only keeps the home copy up to date. To keep the other copies coherent according to the consistency model, a software protocol supplements the hardware support with an invalidation-based scheme. As in LRC, in AURC invalidations are sent to a processor when it does an acquire operation. When a processor which is not the home has a page miss due to the invalidation, the local copy is updated by transferring the entire page from the home on demand (see Figure 9.a). AURC uses *vector timestamps* [19] to partially order the synchronization intervals and the page versions. As in LRC, timestamps are built from local logical counters incremented on each interval, where an interval for a processor is the period between two local synchronization release events.

5.2 AU-based ScC protocol

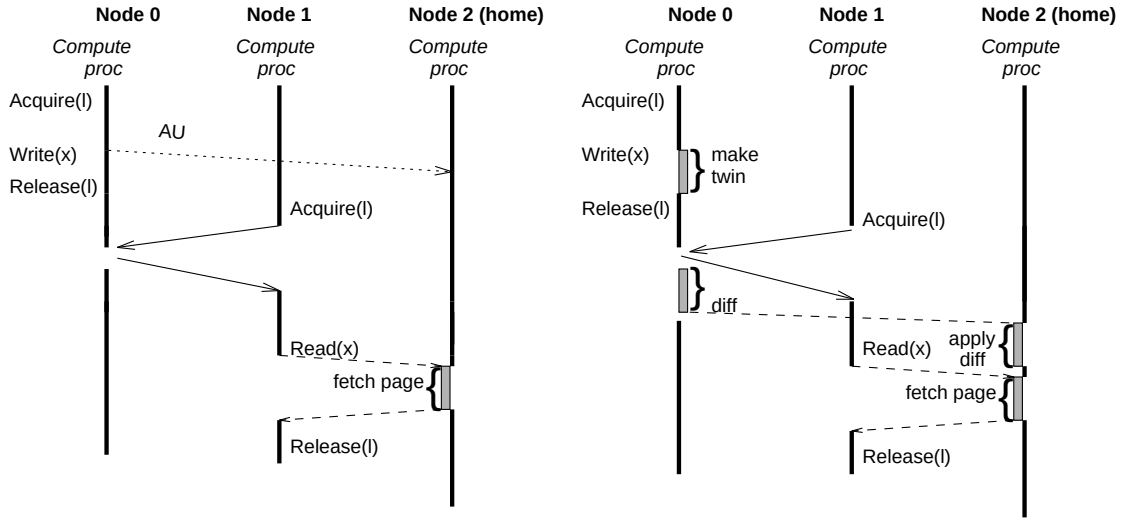
The AU-based scope consistency protocol can be viewed as a per-scope factorization of the AURC protocol. Pages are versioned using vector timestamps as in AURC. In addition, ScC introduces an incarnation number for each lock (scope) which is incremented with each release of that lock (session of that scope). For each lock (scope) incarnation it produces, a processor keeps an *update list* of pages it updated during that incarnation. If a page is updated while several critical sections are open, the update will appear in the update list of all these locks (scopes) for the corresponding incarnations. In particular all pages updated since the last barrier will be reported for the global scope at the next barrier event.

In what follows, we describe how the ScC protocol works for lock-based scopes. The actions are similar for general acquires and releases if explicit scopes are added. Each process remembers the last incarnation it produced for each lock it ever held. At acquire time, the acquiring process sends with the lock request the last incarnation number it produced for that lock. Based on this information, in ScC the releasing processor provides the update lists only for the missing lock incarnations to the acquirer. By contrast, in AURC the releasing processor would have provided the update list for all the missing intervals regardless of whether or not they have been produced by the lock which is acquired. The acquirer uses the update lists to make its address space scope consistent by selectively invalidating the obsolete pages.

5.3 All-software ScC Protocol

The hardware support for automatic update is very valuable for shared virtual memory independent of scope consistency [16, 17], and several shared virtual memory systems have been built using this feature (see [16, 21]). However, ScC can also be built on top of an all-software home-based protocol, without automatic update support. All-software home-based LRC protocols have been shown to perform and scale much better than the traditional, homeless LRC [33].

The home-based LRC protocol(HLRC) [33] is inspired by AURC [16]. The basic scheme consists of using diffs, in the absence of AU support, to detect the updates, propagate them eagerly to the home at the release, and apply them to the home of the page. As a consequence, in HLRC the home copy of the page is kept up-to-date as in AURC but by using diffs and extra software overhead instead of automatic updates (see Figure 9.b). Diffs are not retained at the writers



(a) AURC (b) HLRC

Figure 9: AURC and HLRC Protocols

after they are sent to the home, and they are discarded at home as soon as they are applied, thus greatly reducing storage costs. To satisfy page misses following coherence invalidations, the other nodes will fetch the entire page from the home without any further processing, similar to AURC(alternatively diffs could be propagated to the home, merged when possible but not applied, and diffs not whole pages fetched from the home on a miss). Thus, a page miss does not have to go to multiple writers of the page to get the diffs, as in earlier LRC implementations like TreadMarks [19].

The ScC model can be implemented in an all-software home-based protocol following the same approach used to convert AURC to ScC.

6 Performance

Parameter	Value
Processor clock	60 MHz
Page size	4 Kbytes
Data cache	256 Kbytes
Cache line size	32 bytes
Write buffer size	4 words
Memory bus bandwidth	80 Mbytes/sec
Page transfer bandwidth	28 Mbytes/sec
Trap cost	150 cycles
Message latency	5 μsec

Table 1: System Parameters Used in Simulation.

To evaluate the performance impact of scope consistency, we implemented both the AU-based ScC and AURC protocols on top of a shared virtual memory multiprocessor simulator. The simulator interfaces with the TangoLite execution-driven reference generator [15]. The architectural parameters (Table 1) we use are essentially those of the SHRIMP multicomputer, which has a network interface that supports automatic update. The simulator handles contention in detail both at the memory bus level as well as in the network interface between the incoming and outgoing traffic. Contention in the network itself is not simulated but this doesn't affect the comparison much since the amount of AU-based traffic is the same for both protocols.

Results from previous studies [16, 17] show that AURC substantially outperforms the all-software "homeless" (non home-based) LRC protocol running on the same hardware (the latter does not exploit the automatic update feature).

We evaluate the performance gained from ScC for different classes of sharing patterns in real applications. The programs for this evaluation were selected from the Splash-2 suite [31] to cover distinct cases of interest. The problem sizes used are the default sizes indicated in Splash-2. We also developed a simplified N-body kernel representing a simple pairwise force calculation phase [29], for which we arranged data distribution to cause a lot of false sharing (see description bellow). This is intended to artificially showcase the potential benefits of ScC. We measured the benefits obtained from ScC by two means: reductions in the number of page misses, and increases in speedup (hence performance) obtained. The former measure is objective to a large extent because it is not directly dependent on simulation parameters. The latter is what we ultimately care about.

The simulation results are shown in Figure 10, and will be analyzed shortly. The speedups for an all-software LRC protocol on the same hardware are reported for comparison. In all cases, AURC improves performance substantially compared to the all-software LRC. For applications which are further improved by ScC (Barnes, Raytrace and Cholesky), the number of page misses is reduced from 10% up to 26%. For all three applications, the speedup is improved by 10-14%. Two other applications (LU, Water) are insensitive to ScC. The artificial N-body kernel verifies that the savings in the number of page misses can be dramatic under certain circumstances.

The performance benefit of scope consistency over AURC

Application (problem size)	Number of page misses per processor			Speedup			
	AURC	ScC	% reduction	LRC	AURC	ScC	% improvement
Modified N-body Kernel	1823	1002	45	-	-	-	-
LU (512 x 512)	398	398	0	8.4	9.4	9.4	0
Water-Nsquared (512)	1681	1651	2	8.8	11.3	12	6
Barnes (16 K particles)	2700	2400	11	6	9.2	10.1	10
Raytrace (balls4)	9052	8107	10	6.8	7.3	8.3	14
Cholesky (tk29)	3875	2866	26	3.9	5.8	6.5	12

Figure 10: Simulation Results for 16 Processors

comes from two main sources: less communication due to a reduced false sharing effect, and lower protocol overhead due to simpler coherence handling (coherence bookkeeping is per-lock instead of per-process). We now analyze the sharing patterns of the applications to understand how and when the ScC model provides the savings in the communication traffic. Let us start with the artificial N-body kernel for illustration.

6.1 The Modified N-body kernel

The performance benefit of scope consistency over AURC or other RC based protocols depends on the presence of false sharing effects. Our N-body kernel, a simplified force calculation from an $O(n)$ N-body algorithm using particles and space cells, displays such false sharing. It is based loosely on the Water-spatial application in Splash-2 [31]. The computational domain is divided into cells which are distributed among processors. Each processor computes the interactions of the particles belonging to its cell with particles from nearby cells. As a result of each pairwise interaction between particles, particle properties are modified and particles may move between cells, causing cells to be updated as well. Some global status variables (e.g. temperature and energy) are modified after each interaction.

Every particle and every space cell is protected by a distinct lock. The data structure holding the particles is allocated as an array in this example. Since particles move the locations of particles in the array has little to do with their locations in physical space and hence in cells. Under these circumstances, force computation and particle/cell updates – and particularly, the synchronization for a global status variable – can lead to random false sharing which is difficult to trace. We deliberately choose an order of interactions which causes false sharing to occur once for each pair computation. For instance assume that the i-th interaction at processor p occurs between one of its particles, say a and a particle b which belongs to processor q, and the $i+1$-th interaction at processor q occurs between its particle c and another particle d. The kernel arranges that at the end of i-th interaction, p and q update the global status variables using locks in that order, and also that b and c are located on the same page. The aggressive false sharing that results under LRC or AURC is caused by the lock-based synchronization for global status variables that is unrelated to the particle/cell updates, and is hence perfect for illustrating the gains from ScC. ScC reduces page misses by 45% compared to AURC. Speedups are of course irrelevant for this kernel.

6.2 ScC-insensitive Applications

ScC does not provide any performance improvement for many types of applications. For example, applications that only use barriers and no locks are not helped. The **LU** factorization kernel is an example, and ScC performs exactly like AURC in this case. Another example is **Water-Nsquared**, an $O(n^2)$ molecular dynamics calculation that is another type of N-body simulation. It computes intermolecular interactions using some lock-based synchronization to update particle forces. However, the benefit of ScC is small in this case since the unrelated synchronization effect is not present. Also, accesses to the particle array are very regular and avoid false sharing (there are no space cells, and a processor's assigned particles are allocated contiguously in the particle array).

6.3 ScC-sensitive Applications

The question of whether or not scope consistency can provide any performance benefit on real applications was addressed by choosing three important types of applications: Barnes-Hut, which uses a hierarchical method to solve the N-body problem; Raytrace, a rendering application from computer graphics using ray tracing; and Cholesky, which performs a blocked Cholesky factorization of a sparse positive definite matrix. These applications represent not only distinct domains but also distinct characteristics relevant to ScC.

In **Barnes-Hut** the computational domain is represented as an octree of space cells. The leaves of the octree contain particles, and the computation for both particles and space cells are assigned to processors based on their positions in space. While very different, the computation in some ways follows a similar pattern to that described for our artificial N-body kernel. At each step, the octree is rebuilt based on the current positions of the bodies. Each processor computes the forces for its particles by partially traversing the tree. Interactions with far-away particles are replaced with the interaction with their center of mass. At the end of each time-step the new positions for the bodies are computed.

False sharing occurs at page granularity because the properties of particles and cells are updated in an order that has little to do with their placement in the particle and cell arrays, and the placement in the array has little to do with

the assignment of particles or cells to processes. The false sharing is particularly large in the tree-building phase where many of these updates are protected by per-cell locks, so unrelated lock-protected updates occur by different processes to the same page.

While Barnes-Hut shows performance improvement from ScC, running correctly under ScC required us to expand some critical sections in the tree-building and force calculation phases. We had to make sure that the lock acquire primitive occurs early enough that the critical section includes all the modifications which were supposed to be visible next time the lock is acquired. This is the case, for instance, during the distributed bottom-up computation of the center of mass. Before computing the center of mass for a cell a processor has to wait to have it computed for all of its subcells. When a subcell is ready it signals this through a "done" flag. Before the flag is set various fields of that subcell are updated. Accesses to this done flag occur in critical sections. Under release consistency (and hence AURC), the critical section needs to protect only the flag update: Once the consumer acquires the lock and finds the flag set, it is guaranteed to see all the updates previously performed on that subcell. In ScC, this is not guaranteed unless the critical section is extended to protect the modifications to the cell data structure which are supposed to become visible at the consumer after the flag is set. This is easy to do in the program.

In **Raytrace**, we get about the same benefit as in Barnes-Hut, but with absolutely no change to the program. In fact Raytrace is sensitive to ScC for very different reasons than Barnes-Hut. Raytrace works by creating tasks corresponding to rays that are traced through each pixel in the image plane. Task are dynamically distributed among processors using a system of distributed task queues (one per processor) with task stealing for load balancing. Each task queue is protected by a distinct lock, and the number of lock events is high. Based on the interaction with the read-only scene, the color of each pixel of the image is computed. The image pixel updates are outside any critical section since each pixel is accessed by only the process to which it is assigned. Therefore, there is no need to make the local copies of the image coherent until the next barrier. However, since the image updating alternates with the critical sections for task queue access and management, under AURC or LRC false sharing of image data will induce a lot of unnecessary page invalidations on the image plane, due to the unrelated intervening synchronization. In contrast, in ScC the updates to the image are not in the scope of the task queue locks but in the global scope, so they do not cause invalidations to occur to image pages when acquiring task queue locks(which do not protect the image plane). Updates to image data are collected at the home of the relevant page, and the local copies are invalidated only at the next barrier which closes the global scope. The reduction in false-sharing causes a significant increase in performance, even though most of the time in the program is spent in traversing the read-only scene tracing rays rather than updating the image.

Cholesky is an application which also uses task queues to dynamically distribute computation. The number of task queues is again equal to the number of processors. A task computes one block and produces new tasks corresponding to dependent blocks. These tasks are exported to the task queues of the processors owning those blocks. When a new task executes it assumes that blocks modified by the task that created it are visible. Task queues are accessed with mutual exclusion using locks. The critical sections are small and contain only the insertion and deletion operations on the task queues. However, since the modifications of the blocks themselves occur outside the scope of these critical sections, they are not visible to the new (dependent) tasks and the program does not run correctly under ScC as it is. To ensure correct execution under ScC, the tasks (block updates) themselves need to be contained in scopes related to the corresponding task queue operations. Explicit scope annotations were used to create scopes dynamically and store their identifiers in the task data structure as shown in the example in Figure 6. Introducing explicit scopes increases the protocol overhead due to the communication involved between the scope producer and the scope consumer. The effect is conceptually similar with the increased synchronization cost usually required by EC over LRC. The question is whether the additional cost pays off or not. It turns out that ScC substantially reduces the number of page faults for Cholesky (over 25%). The effects on overall speedup are smaller, about 12%, because of the increase in protocol overhead and communication due to explicit scopes.

7 Related Work

The concept of shared virtual memory was proposed in 1986 [23, 24]. The release consistency model [11] reduces the impact of false sharing in shared virtual memory [8, 7]. New coherence protocols that improve the performance of release consistency in this context include lazy release consistency [20]. The TreadMarks system [19] is an example of state-of-the-art implementations of shared virtual memory on stock hardware. It uses LRC and allows for multiple writers to a page. Entry consistency [3], a different consistency model in which shared data are explicitly associated with some synchronization variable, further reduces the impact of false sharing.

Software-only techniques have been proposed to reduce false sharing itself by providing coherence in software at fine granularity. Blizzard-S [28] and Shasta [26] rewrite an existing executable file to insert a state table lookup before shared-memory references.

Alternative directory-based shared virtual memory protocols for memory mapped network interfaces supporting automatic update as well as hardware remote reads were proposed for Cashmere [21]. The Plus [4], Galactica Net [18], Merlin [25] and its successor SESAME [30] systems implement hardware-based shared memory using a type of write-through mechanism which is similar in some ways to automatic update. The Memory Channel [13] is a network interface similar to SHRIMP which allows remote memory to be mapped into the local virtual address space and have writes propagated automatically, but without a corresponding local memory update.

LRC and EC have been compared in recent studies [32, 1]. Although their results cannot be directly compared with ours, since they compared LRC with EC while we compared AURC with ScC, a few points are worth noting. In the cases when EC wins, the performance improvement over LRC is on average 10-20%. On the other hand, Barnes and Water-Nsquared are reported to perform better for LRC than for EC. The main reason is the loss of prefetching in EC which is not balanced by its reduction of false sharing compared with LRC. But this is exactly the middle ground

that ScC achieves: reduced false sharing using scopes, and prefetching effect due to full-page transfer are provided simultaneously. Therefore ScC reverses the performance comparison for Barnes-Hut and Water-Nsquared in its favor compared to AURC.

8 Conclusions

We have proposed and evaluated a new consistency model, called Scope Consistency, targeted at shared virtual memory systems. It provides the performance benefits of a model like entry consistency, but without the programming complexity of explicitly associating data with synchronization. In most cases, programs that run under RC also run under ScC without modifications, and can achieve performance improvements with no changes. Some programs need changes to comply with ScC, to ensure that semantically related writes are related by synchronization scopes as well. These changes are moving some synchronization operations to expand critical sections or adding explicit scope annotations. We believe that programming in ScC is simpler than in EC since scopes are associated with sections of code rather than with data.

We have also described and evaluated an implementation of scope consistency using the automatic update mechanism provided by recent network interface technologies. We found performance benefits for real applications that suffer false sharing due to unrelated synchronizations, and no losses in performance in other cases. We have also shown how the ScC model can be implemented entirely in software in the absence of automatic update support, by using a home-based protocol. We expect performance benefits in this case as well. While scope consistency is not always completely transparent, we believe it is promising for systems built with the increasingly popular network interfaces that support automatic remote updates, as well as potentially for all-software shared virtual memory systems.

Acknowledgments

We are grateful to the referees, whose comments helped improve the paper. We thank Edward Felten and Cezary Dubnicki for discussions in the early stages of this work and Jim Philbin and Henry Cejtin for providing us with simulation cycles. This project is sponsored in part by ARPA under contract under grant N00014-95-1-1144, by NSF under grant MIP-9420653, by Digital Equipment Corporation and by Intel Corporation.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementation. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[2] S.V. Adve and M.D. Hill. Weak Ordering–A New Definition. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 2–14, June 1990.

[3] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[4] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 115–124, May 1990.

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

[6] Matthias Blumrich, Cezary Dubnick, Edward Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[7] J. B. Carter, J. K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–244, August 1995.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 434–442, June 1986.

[10] E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[12] P.B. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

[13] R. Gillett, M. Collins, and D. Pimm. Overview of Network Memory Channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, February 1996.

[14] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 422–431, June 1988.

[15] S.A. Herrod. *TangoLite; A Multiprocessor Simulation Environment.* Computer Systems Laboratory, Stanford University, 1994.

[16] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[17] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1996.

[18] Andrew W. Wilson Jr. Richard P. LaRowe Jr. and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 246–255, May 1993.

[19] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[20] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[21] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[22] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[23] K. Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[24] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[25] Creve Maples. A High-Performance, Memory-Based Interconnection System For Multicomputer Environments. In *Proceedings of Supercomputing '90*, pages 295–304, November 1990.

[26] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[27] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 234–243, June 1987.

[28] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood. Fine-grain Access for Distributed Shared Memory. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[29] J.P. Singh, J.L. Hennessy, and A. Gupta. Implications of Hierarchical N-Body Methods for Multiprocessor Architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[30] Larry D. Wittie, Gudjon Hermannsson, and Ai Li. Eager Sharing for Efficient Massive Parallelism. In *Proceedings of the 1992 International Conference on Parall el Processing*, pages 251–255, August 1992.

[31] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1995.

[32] M.J. Zekauskas, W.A. Sawdon, , and B.N.Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the Operating Systems Design and Implementation Symposium*, pages 87–100, November 1994.

[33] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.