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

The large granularity of communication and coherence in shared virtual memory systems causes problems with 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 shared virtual memory, called *Scope Consistency* (ScC), which offers most of the potential performance advantages of the EC model without requiring explicit bindings between data and synchronization variables. Instead, ScC dynamically detects the bindings implied by the programmer allowing a programming interface as simple as that of RC or LRC in most cases. We compare ScC with Automatic Update Release Consistency (AURC), a modified LRC protocol which takes advantage of new network interfaces that provide automatic update support. AURC already improves performance substantially over the all-software LRC. For three of the five applications we used, ScC further improves the speedups achieved by AURC by about 10%. We also show how ScC may be used without any hardware support.

1 Introduction

Several relaxed consistency models [7, 6, 14, 2] have been proposed to improve the performance of shared virtual memory systems. These relaxed consistency models alleviate the "false sharing" and extra communication problems caused by using a page as the granularity of communication and coherence. 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) [7] are accepted to offer a reasonable tradeoff between performance and programming complexity for shared virtual memory [5]. RC guarantees memory consistency only at synchronization points, which are marked as acquire or release operations. Updates to shared data are globally performed at each release operation. Lazy Release Consistency (LRC) [14] is a relaxed implementation of RC in which coherence actions are postponed from the release to the next acquire operation. LRC (or RC) 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 [8, 4]. Automatic Update Release Consistency (AURC) [10] is one such protocol that improves performance substantially over all-software LRC, by using the automatic update mechanism provided in the SHRIMP network interface [4]. The additional programming requirements imposed by these 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) [2] lets the acquire operation obtain updates 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 — and thus reduces unnecessary update propagations 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 for shared virtual memory, called **Scope Consistency** (ScC). ScC offers most of the potential performance advantages of EC, without requiring explicit binding of data to synchronization variables. It introduces a concept called a *consistency scope*, which effectively establishes the binding 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. One simple change to make programs ScC is to expand critical sections by moving synchronization operations. However, if the required extension of critical sections is significant, contention for locks may increase and performance may drop. In these cases, the solution is to define additional scopes, either by adding more locks or by using explicit scope annotations. Although explicit scopes add to the programming interface, the extended programming interface for ScC is still much simpler to use than the EC programming interface, since scopes are associated with

To appear in Proceedings of the 8th Anual ACM Symposium on Parallel Algorithms a nd Architectures, June 1996

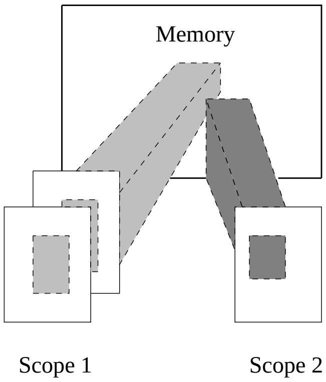

Figure 1: Scope Consistency: The Intuition

sections of code or "tasks" rather than with data. In fact, it often turns out to be easier to add scopes than locks. While ScC as a model is independent of implementation, and can be used with an all-software shared virtual memory protocol as well, to guarantee performance benefits it requires support for fine-grain remote writes (automatic updates). Such support is provided in several recent network interfaces to implement memory-mapped communication (e.g. SHRIMP [4], DEC Memory Channel [8]), and has been found to be very useful for efficiently implementing shared virtual memory [10, 15].

To understand the performance implications of this approach, we implemented ScC and AURC protocols within the TangoLite simulation framework [9]. We conducted detailed simulation studies with five real 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. The other two ScC-improved applications needed program changes to comply with ScC.

2 Scope Consistency

The basic idea in *Scope Consistency* (ScC) is to use a concept called *consistency scope* to implicitly establish the relationship between data and synchronization events, thus realizing a consistency model that is more relaxed than LRC. The main attraction of this approach is that in most cases, consistency scopes can be derived implicitly from the synchronization already present in application programs that conform to Release Consistency. Explicit scopes, when required, are introduced through additional annotations, which are easy to place because the binding is defined in terms of tasks or code sections and not data.

2.1 Consistency Scope

A *consistency scope* is a scope 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, with an additional global consistency scope for barriers which includes the entire program. 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 becomes visible to processes that then enter sessions of that scope (acquire that lock or pass a barrier). Updates made outside the scope session may not be visible. Figure 1 shows the intuition for consistency scopes. Memory is accessed through different scopes that are independent of one another. Each such perspective is built out of sessions which occur on different processors. Sessions belonging to different scopes can interleave in time or overlap in the memory they access. In addition to the individual consistency scopes (whose sessions may be delineated by locks and unlocks, say), there is also a global consistency scope (whose sessions are typically delineated by barriers) with regard to which all memory references are performed.

To define the model more formally, we need the following two definitions that distinguish a reference being *performed with respect to a consistency scope* from a reference being *performed with respect to a processor*:

1. A write that occurs in a consistency scope is *performed with respect to that scope* when the current session of that scope closes.
2. A write is *performed with respect to a processor P* if a subsequent read issued by P returns the value stored by that write.

The consistency rules for scope consistency are:

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

The first rule establishes that after a consistency scope is opened at a processor, the updates previously performed *with respect to that consistency scope* are guaranteed to be performed *with respect to that processor*. In other words, all the updates which occurred in all previously closed sessions of that scope are guaranteed to be performed in the memory local to that processor.

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.

2.2 Lock-based Consistency Scopes

As mentioned earlier, in most programs ordinary synchronization events such as locks and barriers can be used to 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.

P0	P1	
Acquire (L0)	. . . .	--- *open scope L0* ---
X0 = 1		
Acquire (L1)	. . . .	--- *open scope L1* ---
X1 = 1		
Release (L1)	. . . .	--- *close scope L1* ---
Release (L0)	. . . .	--- *close scope L0* ---
	Acquire (L1)	--- *open scope L1* ---
	a = X0	
	b = X1	
	Release (L1)	--- *close scope L1* ---

Figure 2: Programming with Lock-based Scopes

The lock acquire opens the scope session and the lock release closes it. Barriers define a global consistency scope, initially open at all processors. 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. Any benefits from ScC therefore occur through more specific synchronization (e.g. locks) within the region between two barriers.

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.* The first definition can be restated as follows:

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

This definition implies that multiple processors can simultaneously perform writes with respect to a barrier but only one processor at a time can perform writes with respect to a lock. The second definition is as before. The scope consistency rules can be reformulated as following:

1. Before a lock (or barrier) acquire is allow to perform at processor 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 only after all previous acquire events have been performed.

If scope consistency semantics are attached to both locks and barriers, LRC 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 processors. In addition, a memory reference is also considered to occur in all consistency scopes corresponding to the locks which are acquired but not yet released.

Example I

Figure 2 illustrates Scope Consistency with an example. When Process P0 acquires lock L0 it also 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 also exist since the consistency model does not distinguish between synchronizations, 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 updates on X0 and X1 performed by P0 would have become visible at P1.

In most cases, programs that follow the programming discipline of LRC can run with ScC without modifications. However, not all LRC programs are correct under ScC. To run correctly under the ScC model, a LRC program must satisfy the following conditions:

1. Lock-protected modifications to shared data are not expected to be visible at processor P before at least one of the protecting locks is acquired by P.
2. Modifications to shared data that are not protected by a lock, are not expected to be visible at processor P before the next barrier.

Example II

Figure 3 provides an example showing the type of programming problems which can be encountered in switching from LRC to ScC. P_0 updates X first and then Y; Y is updated in the critical section guarded by L_1 lock, while X is updated outside it. P_1 acquires L_1 after P_0 has released it.

In LRC model [14], after acquiring L_1, P_1 is guaranteed to see all writes which occurred at P_0 before the L_1 release, 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 one consistency scope only, namely the one defined by all critical section guarded with the same lock L_1. To guarantee the visibility of both X and Y, the update to X by P_0 must be performed inside the same critical section, which makes sense if the two updates are semantically related. Otherwise, the update on X will be

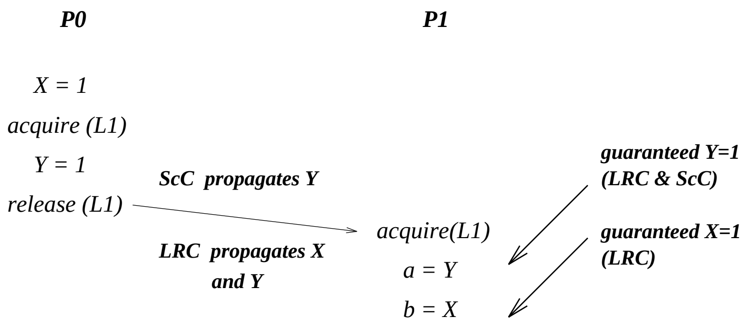

Figure 3: Scope Consistency versus Lazy Release Consistency.

guaranteed visible at P_1 only after either the lock which guarded that update is acquired by P_1, or the next barrier.

Programs which use task queues to distribute the computation exemplify this situation: The critical section protects the task queue update only and not the updates performed by that task. However, the programmer assumes that when the task is selected from the task queue, the updates performed 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 LRC or AURC 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 the critical section 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 which may affect performance significantly. 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.

2.3 Explicit Scopes

Scopes can be created dynamically using explicit annotations. Figure 4 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 incorporate the updates for new tasks and 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 task data structure that is added to the task queue within a critical section (lock-based scope). 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 is modified using explicit scopes to comply with ScC.

Although this approach appears conceptually similar to the use of read-only locks in EC, there are significant differences both in terms of usage model and implementation. Unlike read-only locks in EC, explicit scopes don't require explicit binding of data to locks. They are dynamically created and incorporate all the updates performed while sessions are open. In fact, they correspond to dynamically executed sections of code (and all the memory accesses therein), not to particular data, which makes them much easier to use. Adding more locks is, however, an alternative solution in ScC, and its usage would be very similar to EC except for binding.

3 Comparisons with LRC and EC

Scope Consistency is a relaxed page-based consistency model situated between RC [7] and EC [2] (see also Figure 5).

3.1 ScC versus LRC

Both ScC and LRC are page-based shared virtual memory systems which don't require explicit binding of data to synchronization. However, ScC assumes an implicit binding determined from memory accesses. ScC reduces to LRC for applications that use only global synchronization, such as barriers, because 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 updates 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 seeing memory updates until the corresponding scope is open. This allow the acquiring processor to perform (through invalidations, say) only the updates which occurred in the same scope.

Figure 6 is an example showing 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 (on X_0 and X_2) since the acquire event is allowed to perform only after all previous memory accesses

```
        P0                    P1

S = open_scope()   . . . . . . . . . . . . . . . . . . . .   --- create a new scope ---
   X0 = 1
close_scope (S)    . . . . . . . . . . . . . . . . . . . .   --- close the new scope ---
Acquire (L)        . . . . . . . . . . . . . . . . . . . .   --- open scope L ---
   task. scope = S . . . . . . . . . . . . . . . . . . . .   --- save the scope id ---
Release (L)        . . . . . . . . . . . . . . . . . . . .   --- close scope L ---
                        Acquire (L)            . . . . . .   --- open scope L ---
                          T = task.scope       . . . . . .   --- get the scope id in T ---
                        Release (L)            . . . . . .   --- close scope L ---
                        open_scope (T)         . . . . . .   --- open scope T ---
                          a = X0
```

Figure 4: Programming with Explicit Scopes

	LRC	ScC	EC
Association between data and synchronizations	-	Implicit	Explicit
Coherence granularity	Page	Page	Object
Programming compared to LRC	-	Same or Extended	Different

Figure 5: Scope Consistency Characterization

have been performed. Therefore, both pages (*0* and *1*) will be invalidated in an invalidation-based protocol.

In fact, P_1 does not need to see the update 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_0 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.

Hardware support LRC is a consistency model which can be implemented completely in software using different all-software write collection schemes [1]. AURC is a variant of LRC which takes advantage of automatic update support provided in the new network interfaces to merge the updates from multiple writers [10].

ScC needs the same kind of hardware support as AURC in order to guarantee its performance benefit. The reason is that while ScC postpones some page invalidations, all updates must be eventually performed at the next barrier (global scope) if not earlier. Consequently, ScC is effective only if reduction in page faults between barriers translates to savings in communication cost. This happens if updates are merged so that the cost of obtaining an updated copy of a page after a barrier is less than the cost saved by avoiding obtaining the updates between barriers. This is the case in AU-based protocols because obtaining an updated page is done by fetching it from the home with no additional processing. Therefore, by reducing the number of page invalidations the update cost is also proportionally reduced. A diff-based all software LRC scheme like the one used in TreadMarks[13] doesn't have the same property of keeping a home copy up-to-date. Updates are embedded in *diffs* which have to be transferred eventually and applied on every local copy. Therefore, postponing some invalidations and saving some page faults doesn't ultimately save communication cost. An all-software scheme with this property can be constructed by combining the ideas of TreadMarks and AURC, i.e. propagating modifications automatically to a home copy in software. This allows ScC to be implemented entirely in software, although the hybrid LRC scheme can be less effective than standard LRC and can therefore compromise the role of the ScC model.

Programming precaution Like LRC and all-software systems, ScC requires explicit synchronizations which must be labeled with system-supplied primitives (e.g. lock, barrier).

ScC delineates a scope by all critical sections protected by a given lock. However, since only the updates performed within the scope sessions are guaranteed to be performed in that scope, not all programs correct under LRC are guaranteed to be correct under ScC, as we have seen. Although program modifications are sometimes necessary to ensure the correctness, they are much simpler to perform than for EC.

3.2 ScC versus EC

Both EC and ScC are consistency models defined for shared memory. Both schemes relax the memory consistency model by taking advantage of the relationship between data and synchronization. Both schemes are particularly effective for applications based on extensive use of point-to-point (lock) synchronization. However the solutions proposed by the two schemes are quite different.

Binding. In EC [2], the binding between *data objects* and *synchronization objects* is specified explicitly by the programmer. In fact, the solution proposed by EC goes beyond page-based shared memory systems, approaching object-based or region-based shared memory models. EC requires explicit associations of data with synchronizations because the protocol relies on this information to identify which updates must be propagated and when. In a system

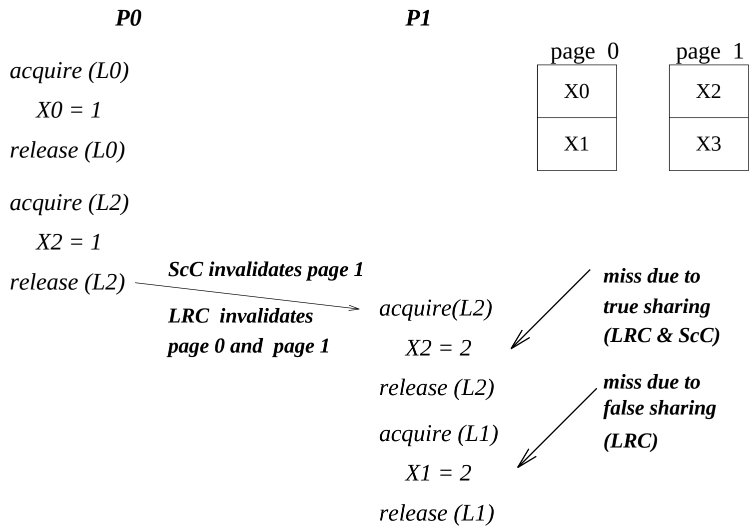

Figure 6: False Sharing in ScC versus LRC

like Midway [2] which supports both the EC and RC models, the default model is RC. Only the data objects explicitly associated with synchronization variables can be treated using the EC. In addition, in may cases the binding may change during the program lifetime, thus requiring 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 updates which occur in the sessions of that scope. We can say that EC binds data explicitly to locks while the ScC binds dynamic memory references implicitly to locks.

Global Synchronization. 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. 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. The second approach is difficult to implement and may even be impossible since it may require the programmer to anticipate and specify the different bindings for the same synchronization object. 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. However, since ScC employs an invalidation-based protocol, this solution is reasonable.

Granularity There is 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. The communication granularity depends on how write collection is implemented. Midway [2] uses software dirty bits, so the amount of communication is proportional to the size of the updated data (not like diffs in LRC).

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 still of page size which means that in order to update a location which became bound to a scope, the entire page which contains that location is sent over. On the other hand, although the communication volume may be higher in ScC than in EC due to page fragmentation, the page size transfer in ScC allows prefetching to occur [1].

Let us now examine how ScC might be implemented using the automatic update support exploited by the AURC protocol. Implementing all-software ScC will be discussed in Section 6.

4 Implementation of ScC with AU

We first describe the hardware support called automatic update, and then a shared virtual memory protocol which uses the *automatic update* support to implement scope consistency.

4.1 Automatic Update

Automatic update can be viewed as a write-through between two local memories. It allows selective writes 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

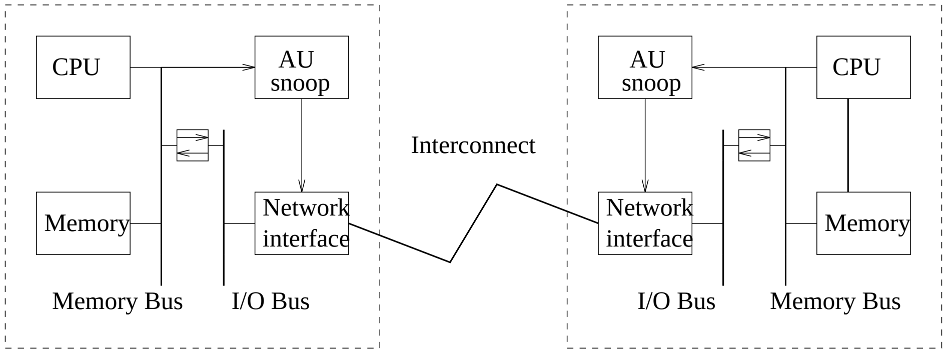

Figure 7: Block Diagram of the SHRIMP system

page B at node 1. The network interface of the SHRIMP multicomputer implements automatic update by snooping all write traffic on the local memory bus (see Figure 7). Writes to local memory pages which are also mapped out are intercepted and forwarded to the remote destination through the I/O bus. This AU feature provides adequate support for a shared virtual memory implementation. AURC is such a protocol designed to support LRC in this environment. We now briefly describe the AURC protocol. For details see [10].

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 replaces more expensive all-software solutions for write detection used in LRC implementations, like diff computation [13] or software dirty bit schemes [23]. This mechanism only keeps the home copy up to date. To keep the other copies coherent according to the consistency model, a software protocol must supplement the hardware support with an invalidation-based scheme. As in LRC, invalidations are sent to a processor when it does an acquire operation. Then, 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. AURC uses *vector timestamps* [13] to describe synchronization intervals and the page version. As in LRC, timestamps are built from local counters incremented on each interval, where an interval for a processor is the period between two local synchronization release events. Timestamping allows selective page invalidation and implicit prefetch detection. Selective invalidation means that only those pages whose timestamps indicate that they are not up-to-date are invalidated. Implicit prefetching occurs when the version of the page fetched on demand from the home is more recent than the one requested. By remembering its version, prefetching can be detected to avoid further invalidations.

When a page is fetched from the home, the protocol must guarantee that the "previous" automatic updates from other processors are already in place at the home. This is accomplished by using the timestamp to flush the automatic update data links according to the coherence protocol. Such a flushing scheme also allows the home to determine the version of page it possess at any instant of time.

4.2 The ScC protocol

The 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, along with the interval when each page was updated (recall that an interval is the period between two consecutive scope session endings or releases at the same processor). If a page is updated while several locks are held (scope sessions are open), it will appear in the update list of all these locks (scopes). In particular all pages updated since the last barrier will be reported at the next barrier event.

In what follows, we describe how the ScC protocol works for lock-based scopes. The actions are similar for explicit scopes. Each process remembers the last incarnation it produced for each lock. When a specific lock is acquired the acquiring processor sends with its lock request the last incarnation number it knows of for that lock. Based on this information, the releasing processor provides the update lists for the missing lock incarnations to the acquirer, and the acquirer invalidates the corresponding pages. This scheme guarantees that only the writes which have been performed with respect to that lock are performed at the acquirer. In fact, coherence is much simpler to manage in ScC then in AURC. The difference is that in AURC the acquiring processor presents to the releasing processor its vector timestamp and requests full information about *all* intervals which are not reflected in that timestamp, while ScC is interested only for the missing intervals which occurred in the same scope.

5 Performance

To evaluate the performance impact of Scope Consistency, we implemented both ScC-based and AURC protocols on top of a shared virtual memory multiprocessor simulator. The simulator interfaces with the TangoLite execution-driven reference generator [9]. 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

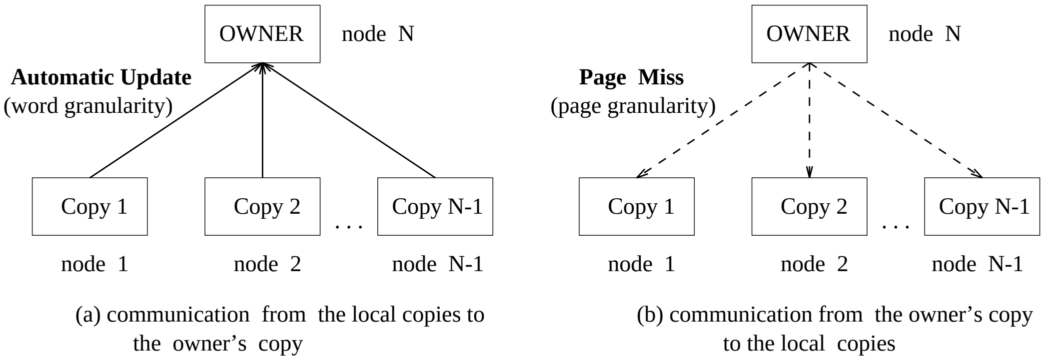

Figure 8: Basic Communication Schemes in AURC

the network itself is not simulated.

We compare the performance of ScC against AURC as implemented on a SHRIMP-like machine. Results from a previous studies [10, 11] show that AURC substantially outperforms the all-software LRC protocol running on the same hardware (the latter does not exploit the automatic update feature).

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 send latency	5 μsec

Table 1: System Parameters Used in Simulation.

We evaluate the performance gained from ScC for different classes of sharing patterns in real applications. The programs for this evaluation were appropriately selected from the Splash-2 suite [22] to cover distinct cases of interest. The problem sizes used are the default sizes indicated in Splash2. We also developed a simplified N-body kernel representing a simple pairwise force calculation phase [20], for which we arranged data distribution to cause a lot of false sharing. This is intended to to illustrate a situation which showcases 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 9, 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. And 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 comes from two main sources: less communication due to a reduced false sharing effect, and lower protocol overhead due to simpler coherence handling. 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.

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 [22]. 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 also 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. 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 can lead to random false sharing which is difficult to trace. We induce a lot of false sharing in our kernel by deliberately choosing 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 m which belongs to processor r, and the i+1 interaction at processor r occurs between its particle n and another particle x. Then the kernel arranges that at the end of i-th interaction, p and r update the global status variable in that order, and also that m and n 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 irrelevant for this kernel.

ScC-insensitive Applications ScC doesn't provide any

Application (problem size)	Number of page misses per processor			Speedup			
	AURC	ScC	%	LRC	AURC	ScC	%
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 9: Simulation Results for 16 Processors

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 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).

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 both particles and space cells are distributed among 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 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 step the new positions for the bodies are computed. Properties of particles and cells are updated in an order that has little to do with their placement in the particle and cell arrays (hence causing false sharing at page granularity), and particularly in the tree-building phase many of these updates are protected by per-cell locks.

While Barnes-Hut shows performance improvement from ScC, running 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 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 for 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 one processor. 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. 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 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 queue 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. 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. Since the block computation occurs outside the scope of these critical sections, the program doesn't 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 4. 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 diminished to about 12% because of the increase in protocol overhead and communication.

6 All-software Implementation for ScC

The hardware support for automatic update is very valuable for shared virtual memory independent of scope consistency [10], and several shared virtual memory systems have been built using this feature (see [10, 15, 11]). We have seen that scope consistency can be easily built on top of this in software, and has performance benefits. However, it is also interesting to see if ScC can be built on top of an all-software protocol, without automatic update support. One way to do this is to emulate the AURC protocol completely in software, using diff-based write detection as in LRC. However, instead of sending and applying diffs on demand as in LRC, the software ScC protocol will send the diffs eagerly to a designated home node, where these diffs are immediately applied on the home copy. As a consequence, the home copy of the page is continuously updated as in AURC but by using diffs and extra software overhead instead of automatic updates. To satisfy page misses following coherence invalidations, the other nodes will fetch the entire page from the home without any further processing, as in AURC. The ScC model can be implemented starting from this software AURC emulation protocol as described earlier.

The main question regarding this hybrid protocol is performance. Although we have seen about 10% performance improvement due to ScC, the efficiency of software-emulated AURC compared with the alternative LRC is difficult to predict and most likely depends on both applications and architectures. Compared with LRC, the AURC emulation will save a number of page faults (at home), the protocol is simpler, diffs are applied only once (at home) and misses are solved with no more than one round-trip (to the home). But at the same time, unlike in LRC, in the AURC emulation misses are always satisfied by fetching all pages even for small changes. Thus, for applications with medium to coarse-grained, one producer-multiple consumer sharing patterns (like LU or Cholesky), the software AURC emulation is likely to win over LRC, while for applications with fine-grained migratory sharing patterns the reverse is likely to be true. The comparison will also be influenced by the node configuration (uniprocessor or SMP, with or without a processor reserved for protocol processing). It is difficult to speak of the performance improvement due to ScC in this context except by comparing with the original LRC.

7 Related Work

The concept of shared virtual memory was proposed in Kai Li's Ph.D. thesis in 1986 [16, 17]. Relaxed consistency models [7] allow shared virtual memory to reduce the cost of false sharing. New coherence protocols that improve the performance of release consistency in this context include lazy release consistency [14], and entry consistency [2], in which shared data are explicitly associated with some synchronization variable.

The TreadMarks library [13] 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. This implementation provides respectable performance in the absence of fine-grained sharing.

Software-only techniques have been proposed to reduce false sharing by controlling fine-grain accesses to shared memory. Blizzard-S [19] rewrites an existing executable file to insert a state table lookup before every shared-memory reference. This technique works well in the presence of fine-grain sharing.

Alternative directory-based protocols for memory mapped network interfaces were proposed for Cashmere [15]. The Plus [3], Galactica Net [12], Merlin [18] and its successor SESAME [21], systems implement hardware-based shared memory using a sort of write-through mechanism which is similar in some ways to automatic update. The Memory Channel [8] allows remote memory to be mapped into the local virtual address space and have writes propagated automatically, but without a corresponding local memory mapping.

LRC and EC have been compared in recent studies [23, 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 % which is comparable with the results we reported comparing AURC with ScC. On the other hand, Barnes and Water are reported to perform better for LRC than for EC. The main reason is that the loss of prefetching in going to EC is not balanced by the reduction in false sharing. But this is exactly the middle ground that ScC achieves (since whole pages are fetched in this case, providing prefetching). Therefore ScC reverses the comparison for both applications in the favor of the more relaxed model.

8 Conclusions

We have proposed and evaluated 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, the programs that run under LRC also run under ScC without modifications. Some programs need changes to comply with ScC, to ensure that semantically related writes are related by synchronization scopes as well, but they are usually as simple as moving some synchronization operations to expand critical sections thus preserving the same programming interface as LRC. Even when explicit scope annotations are used, programming in ScC is much simpler than in EC since scopes are associated with sections of code or "tasks" 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. We believe that Scope Consistency 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 using symmetric multiprocessors (SMPs) rather than uniprocessors as nodes, but the performance gains over a wide range of applications using different architectural solutions, need further evaluation.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Richard Gillett. Memory Channel Network for PCI. In *Proceedings of Hot Interconnects '95 Symposium*, August 1995.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.