Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons,
Anoop Gupta, and John Hennessy

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Scalable shared-memory multiprocessors distribute memory among the processors and use scalable interconnection networks to provide high bandwidth and low latency communication. In addition, memory accesses are cached, buffered, and pipelined to bridge the gap between the slow shared memory and the fast processors. Unless carefully controlled, such architectural optimizations can cause memory accesses to be executed in an order different from what the programmer expects. The set of allowable memory access orderings forms the memory consistency model or event ordering model for an architecture.

This paper introduces a new model of memory consistency, called *release consistency*, that allows for more buffering and pipelining than previously proposed models. A framework for classifying shared accesses and reasoning about event ordering is developed. The release consistency model is shown to be equivalent to the sequential consistency model for parallel programs with sufficient synchronization. Possible performance gains from the less strict constraints of the release consistency model are explored. Finally, practical implementation issues are discussed, concentrating on issues relevant to scalable architectures.

1 Introduction

Serial computers present a simple and intuitive model of the memory system to the programmer. A load operation returns the last value written to a given memory location. Likewise, a store operation binds the value that will be returned by subsequent loads until the next store to the same location. This simple model lends itself to efficient implementations—current uniprocessors use caches, write buffers, interleaved main memory, and exploit pipelining techniques. The accesses may even be issued and completed out of order as long as the hardware and compiler ensure that data and control dependences are respected.

For multiprocessors, however, neither the memory system model nor the implementation is as straightforward. The memory system model is more complex because the definitions of "last value written", "subsequent loads", and "next store" become unclear when there are multiple processors reading and writing a location. Furthermore, the order in which shared memory operations are done by one process may be used by other processes to achieve implicit synchronization. For example, a process may set a flag variable to indicate that a data structure it was manipulating earlier is now in a consistent state. Consistency models place specific requirements on the order that shared memory accesses (*events*) from one process may be observed by other processes in the machine. More generally, the consistency model specifies what event orderings are legal when several processes are accessing a common set of locations.

Several memory consistency models have been proposed in the literature: examples include sequential consistency [7], processor consistency [5], and weak consistency [4]. The *sequential consistency* model [7] requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. While conceptually simple, the sequential consistency model imposes severe restrictions on the outstanding accesses that a process may have and effectively prohibits many hardware optimizations that could increase performance. Other models attempt to relax the constraints on the allowable event orderings, while still providing a reasonable programming model for the programmer.

Architectural optimizations that reduce memory latency are especially important for scalable multiprocessor architectures. As a result of the distributed memory and general interconnection networks used by such multiprocessors [8, 9, 12], requests issued by a processor to distinct memory modules may execute out of order. Caching of data further complicates the ordering of accesses by introducing multiple copies of the same location. While memory accesses are atomic in systems with a single copy of data (a new data value becomes visible to all processors at the same time), such atomicity may not be present in cache-based systems. The lack of atomicity introduces extra complexity in implementing consistency models. A system architect must balance the design by providing a memory consistency model that allows for high performance implementations and is acceptable to the program-

mer.

In this paper, we present a new consistency model called *release consistency*, which extends the weak consistency model [4] by utilizing additional information about shared accesses. Section 2 presents a brief overview of previously proposed consistency models. The motivation and framework for release consistency is presented in Section 3. Section 4 considers equivalences among the several models given proper information about shared accesses. Section 5 discusses potential performance gains for the models with relaxed constraints. Finally, Section 6 discusses implementation issues, focusing on issues relevant to scalable architectures.

2 Previously Proposed Memory Consistency Models

In this section, we present event ordering requirements for supporting the sequential, processor, and weak consistency models. Although the models discussed in this section have already been presented in the literature, we discuss them here for purposes of completeness, uniformity in terminology, and later comparison. Readers familiar with the first three models and the event ordering terminology may wish to skip to Section 3.

To facilitate the description of different event orderings, we present formal definitions for the stages that a memory request goes through. The following two definitions are from Dubois *et al.* [4, 10]. In the following, P_i refers to processor i.

> **Definition 2.1: Performing a Memory Request**
> A LOAD by P_i is considered *performed with respect to* P_k at a point in time when the issuing of a STORE to the same address by P_k cannot affect the value returned by the LOAD. A STORE by P_i is considered *performed with respect to* P_k at a point in time when an issued LOAD to the same address by P_k returns the value defined by this STORE (or a subsequent STORE to the same location). An access is *performed* when it is performed with respect to all processors.

Definition 2.2 describes the notion of *globally performed* for LOADs.

> **Definition 2.2: Performing a LOAD Globally**
> A LOAD is *globally performed* if it is performed *and* if the STORE that is the source of the returned value has been performed.

The distinction between performed and globally performed LOAD accesses is only present in architectures with non-atomic STOREs. A STORE is atomic if the value stored becomes readable to all processors at the same time. In architectures with caches and general interconnection networks, a STORE operation is inherently non-atomic unless special hardware mechanisms are employed to assure atomicity.

From this point on, we implicitly assume that uniprocessor control and data dependences are respected. In addition, we assume that memory is kept coherent, that is, all writes to the same location are serialized in some order and are performed in that order with respect to any processor. We have formulated the conditions for satisfying each model such that a process needs to keep track of only requests initiated by itself. Thus, the compiler and hardware can enforce ordering on a per process(or) basis. We define *program order* as the order in which accesses occur in an execution of the single process given that no reordering takes place. When we use the phrase *"all previous accesses"*, we mean all accesses in the program order that are before the current access. In presenting the event ordering conditions to satisfy each model, we assume that the implementation avoids deadlock by ensuring that accesses that occur previously in program order eventually get performed (globally performed).

2.1 Sequential Consistency

Lamport [7] defines *sequential consistency* as follows.

> **Definition 2.3: Sequential Consistency**
> A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Scheurich and Dubois [10, 11] have described event order restrictions that guarantee sequential consistency. Condition 2.1 presents sufficient conditions for providing sequential consistency (these differ slightly from conditions given in [10]).

> **Condition 2.1: Sufficient Conditions for Sequential Consistency**
> (A) before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be *globally* performed and all previous STORE accesses must be performed, and
> (B) before a STORE is allowed to perform with respect to any other processor, all previous LOAD accesses must be *globally* performed and all previous STORE accesses must be performed.

2.2 Processor Consistency

To relax some of the orderings imposed by sequential consistency, Goodman introduces the concept of *processor consistency* [5]. Processor consistency requires that writes issued from a processor may not be observed in any order other than that in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. Processor consistency is weaker than sequential consistency; therefore, it may not yield 'correct' execution if the programmer assumes sequential consistency. However, Goodman claims that most applications give the

same results under the processor and sequential consistency models. Specifically, he relies on programmers to use explicit synchronization rather than depending on the memory system to guarantee strict event ordering. Goodman also points out that many existing multiprocessors (e.g., VAX 8800) satisfy processor consistency, but do not satisfy sequential consistency.

The description given in [5] does not specify the ordering of read accesses completely. We have defined the following conditions for processor consistency.

Condition 2.2: Conditions for Processor Consistency
(A) before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed, and
(B) before a STORE is allowed to perform with respect to any other processor, all previous accesses (LOADs and STOREs) must be performed.

The above conditions allow reads following a write to bypass the write. To avoid deadlock, the implementation should guarantee that a write that appears previously in program order will eventually perform.

2.3 Weak Consistency

A weaker consistency model can be derived by relating memory request ordering to synchronization points in the program. As an example, consider a processor updating a data structure within a critical section. If the computation requires several STORE accesses and the system is sequentially consistent, then each STORE will have to be delayed until the previous STORE is complete. But such delays are unnecessary because the programmer has already made sure that no other process can rely on that data structure being consistent until the critical section is exited. Given that all synchronization points are identified, we need only ensure that the memory is consistent at those points. This scheme has the advantage of providing the user with a reasonable programming model, while permitting multiple memory accesses to be pipelined. The disadvantage is that all synchronization accesses must be identified by the programmer or compiler.

The *weak consistency* model proposed by Dubois *et al.* [4] is based on the above idea. They distinguish between ordinary shared accesses and synchronization accesses, where the latter are used to control concurrency between several processes and to maintain the integrity of ordinary shared data. The conditions to ensure weak consistency are given below (slightly different from the conditions given in [4]).

Condition 2.3: Conditions for Weak Consistency
(A) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous *synchronization* accesses must be performed, and
(B) before a *synchronization* access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and
(C) *synchronization* accesses are sequentially consistent with respect to one another.

3 The Release Consistency Model

This section presents the framework for release consistency. There are two main issues explored in this section—performance and correctness. For performance, the goal is to exploit additional information about shared accesses to develop a memory consistency model that allows for more efficient implementations. Section 3.1 discusses a categorization of shared accesses that provides such information. For correctness, the goal is to develop weaker models that are equivalent to the stricter models as far as the results of programs are concerned. Section 3.2 introduces the notion of properly-labeled programs that is later used to prove equivalences among models. Finally, Section 3.3 presents the release consistency model and discusses how it exploits the extra information about accesses.

3.1 Categorization of Shared Memory Accesses

We first describe the notions of *conflicting accesses* (as presented in [13]) and *competing accesses*. Two accesses are conflicting if they are to the same memory location and at least one of the accesses is a STORE.[1] Consider a pair of conflicting accesses a_1 and a_2 on different processors. If the two accesses are not ordered, they may execute simultaneously thus causing a race condition. Such accesses a_1 and a_2 form a *competing pair*. If an access is involved in a competing pair under any execution, then the access is considered a *competing access*.

A parallel program consisting of individual processes specifies the actions for each process and the interactions among processes. These interactions are coordinated through accesses to shared memory. For example, a producer process may set a flag variable to indicate to the consumer process that a data record is ready. Similarly, processes may enclose all updates to a shared data structure within lock and unlock operations to prevent simultaneous access. All such accesses used to enforce an ordering among processes are called *synchronization accesses*. Synchronization accesses have two distinctive characteristics: (i) they are competing accesses, with one process writing a variable and the other reading it; and (ii) they are frequently used to order conflicting accesses (i.e.,

[1] A read-modify-write operation can be treated as an atomic access consisting of both a load and a store.

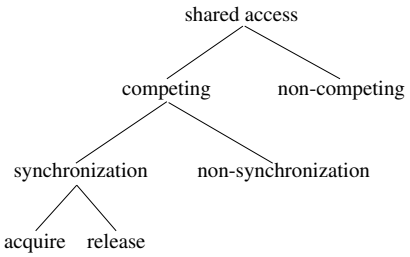

Figure 1: Categorization of shared writable accesses.

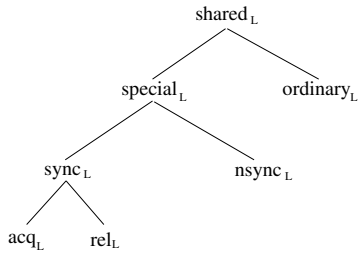

Figure 2: Labels for memory accesses.

make them non-competing). For example, the lock and unlock synchronization operations are used to order the non-competing accesses made inside a critical section.

Synchronization accesses can further be partitioned into *acquire* and *release* accesses. An acquire synchronization access (e.g., a lock operation or a process spinning for a flag to be set) is performed to gain access to a set of shared locations. A release synchronization access (e.g., an unlock operation or a process setting a flag) grants this permission. An acquire is accomplished by reading a shared location until an appropriate value is read. Thus, an acquire is always associated with a read synchronization access (atomic read-modify-write accesses are discussed in Section 3.2). Similarly, a release is always associated with a write synchronization access.

Not all competing accesses are used as synchronization accesses, however. As an example, programs that use chaotic relaxation algorithms make many competing accesses to read their neighbors' data. However, these accesses are not used to impose an ordering among the parallel processes and are thus considered *non-synchronization* competing accesses in our terminology. Figure 1 shows this categorization for memory accesses.

The categorization of shared accesses into the suggested groups allows one to provide more efficient implementations by using this information to relax the event ordering restrictions. For example, the purpose of a release access is to inform other processes that accesses that appear before it in program order have completed. On the other hand, the purpose of an acquire access is to delay future access to data until informed by another process. The categorization described here can be extended to include other useful information about accesses. The tradeoff is how easily that extra information can be obtained from the compiler or the programmer and what incremental performance benefits it can provide.

Finally, the method for identifying an access as a competing access depends on the consistency model. For example, it is possible for an access to be competing under processor consistency and non-competing under sequential consistency. While identifying competing pairs is difficult in general, the following conceptual method may be used under sequential consistency. Two conflicting accesses b_1 and b_2 on different processes form a competing pair if there exists at least one legal interleaving where b_1 and b_2 are *adjacent*.

3.2 Properly-Labeled Programs

The previous subsection described a categorization based on the intrinsic properties of an access. We now describe the labelings for an access. The label represents what is asserted about the categorization of the access. It is the responsibility of the compiler or the programmer to provide labels for the accesses. Figure 2 shows possible labelings for memory accesses in a program. The labels shown correspond to the categorization of accesses depicted in Figure 1. The subscript L denotes that these are labels. The labels at the same level are disjoint, and a label at a leaf implies all its parent labels.

The release consistency model exploits the information conveyed by the labels to provide less strict event ordering constraints. Thus, the labels need to have a proper relationship to the actual category of an accesses to ensure correctness under release consistency. For example, the $ordinary_L$ label asserts that an access is non-competing. Since the hardware may exploit the $ordinary_L$ label to use less strict event orderings, it is important that the $ordinary_L$ label be used only for non-competing accesses. However, a non-competing access can be conservatively labeled as $special_L$. In addition, it is important that *enough* competing accesses be labeled as acq_L and rel_L to ensure that the accesses labeled $ordinary_L$ are indeed non-competing. The following definition provides a conceptual model for determining whether enough $special_L$ accesses have been categorized as $sync_L$ (again assuming the sequential consistency model).

Definition 3.1: Enough $Sync_L$ Labels
Pick any two accesses u on processor P_u and v on processor P_v (P_u not the same as P_v) such that the two accesses conflict, and at least one is labeled as $ordinary_L$. Under any legal interleaving, if v appears after (before) u, then there needs to be at least one $sync_L$ write (read) access on P_u and one $sync_L$ read (write) on P_v separating u and v, such that the write appears before the read. There are *enough* accesses labeled as $sync_L$ if the above condition holds for all possible pairs u and v. A $sync_L$ read has to be labeled as acq_L and a $sync_L$ write has to be labeled as rel_L.

To determine whether all labels are appropriate, we present the notion of properly-labeled programs.

Definition 3.2: Properly-Labeled (PL) Programs
A program is *properly-labeled (PL)* if the following hold: $(shared\ access) \subseteq shared_L$, $competing \subseteq special_L$, and *enough* (as defined above) $special_L$ accesses are labeled as acq_L and rel_L.

An acq_L or rel_L label implies the $sync_L$ label. Any $special_L$ access that is not labeled as $sync_L$ is labeled as $nsync_L$. In addition, any $shared_L$ access that is not labeled as $special_L$ is labeled as $ordinary_L$. Note that this categorization is based on access and not on location. For example, it is possible that of two accesses to the same location, one is labeled $special_L$ while the other is labeled $ordinary_L$.

Most architectures provide atomic read-modify-write operations for efficiently dealing with competing accesses. The load and store access in the operation can be labeled separately based on their categorization, similarly to individual load and store accesses. The most common label for a read-modify-write is an acq_L for the load and an $nsync_L$ for the store. A prevalent example of this is an atomic test-and-set operation used to gain exclusive access to a set of data. Although the store access is necessary to ensure mutual exclusion, it does not function as either an acquire or a release. If the programmer or compiler cannot categorize the read-modify-write appropriately, the conservative label for guaranteeing correctness is acq_L and rel_L for the load and store respectively (the operation is treated as both an acquire and a release).

There is no unique labeling to make a program a PL program. As long as the above subset properties are respected, the program will be considered properly-labeled. Proper labeling is not an inherent property of the program, but simply a property of the labels. Therefore, any program can be properly labeled. However, the less conservative the labeling, the higher is the potential for performance benefits.

Given perfect information about the category of an access, the access can be easily labeled to provide a PL program. However, perfect information may not be available at all times. Proper labeling can still be provided by being conservative. This is illustrated in the three possible labeling strategies enumerated below (from conservative to aggressive). Only leaf labels shown in Figure 2 are discussed (remember that a leaf label implies all parent labels).

1. If competing and non-competing accesses can not be distinguished, then all reads can be labeled as acq_L and all writes can be labeled as rel_L.

2. If competing accesses can be distinguished from non-competing accesses, but synchronization and non-synchronization accesses can not be distinguished, then all accesses distinguished as non-competing can be labeled as $ordinary_L$ and all competing accesses are labeled as acq_L and rel_L (as before).

3. If competing and non-competing accesses are distinguished and synchronization and non-synchronization accesses are distinguished, then all non-competing accesses can be labeled as $ordinary_L$, all non-synchronization accesses can be labeled as $nsync_L$, and all synchronization accesses are labeled as acq_L and rel_L (as before).

We discuss two practical ways for labeling accesses to provide PL programs. The first involves parallelizing compilers that generate parallel code from sequential programs. Since the compiler does the parallelization, the information about which accesses are competing and which accesses are used for synchronization is known to the compiler and can be used to label the accesses properly.

The second way of producing PL programs is to use a programming methodology that lends itself to proper labeling. For example, a large class of programs are written such that accesses to shared data are protected within critical sections. Such programs are called *synchronized programs*, whereby writes to shared locations are done in a mutually exclusive manner (no other reads or writes can occur simultaneously). In a synchronized program, all accesses (except accesses that are part of the synchronization constructs) can be labeled as $ordinary_L$. In addition, since synchronization constructs are predefined, the accesses within them can be labeled properly when the constructs are first implemented. For this labeling to be proper, the programmer must ensure that the program is synchronized.

Given a program is properly-labeled, the remaining issue is whether the consistency model exploits the extra information conveyed by the labels. The sequential and processor consistency models ignore all labels aside from $shared_L$. The weak consistency model ignores any labelings past $ordinary_L$ and $special_L$. In weak consistency, an access labeled $special_L$ is treated as a synchronization access and as both an acquire and a release. In contrast, the release consistency model presented in the next subsection exploits the information conveyed by the labels at the leaves of the labeling tree.

From this point on, we do not distinguish between the categorization and the labeling of an access, unless this distinction is necessary.

3.3 Release Consistency

Release consistency is an extension of weak consistency that exploits the information about acquire, release, and non-synchronization accesses. The following gives the conditions for ensuring *release consistency*.

Condition 3.1: Conditions for Release Consistency (A) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed, and (B) before a *release* access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and (C) *special accesses* are processor consistent with respect to one another.

Four of the ordering restrictions in weak consistency are not present in release consistency. The first is that ordinary LOAD and STORE accesses following a release access do not have to be delayed for the release to complete; the purpose of the release synchronization access is to signal that previous accesses in a critical section are complete, and it does not have anything to say about ordering of accesses following it. Of course, the local dependences within the same processor must still be respected. Second, an acquire synchronization access need not be delayed for previous ordinary LOAD and STORE accesses to be performed. Since an acquire access is not giving permission to any other process to read/write the previous pending locations, there is no reason for the acquire to wait for them to complete. Third, a non-synchronization special access does not wait for previous ordinary accesses and does not delay future ordinary accesses; a non-synchronization access does not interact with ordinary accesses. The fourth difference arises from the ordering of special accesses. In release consistency, they are only required to be processor consistent and not sequentially consistent. For all applications that we have encountered, sequential consistency and processor consistency (for special accesses) give the same results. Section 4 outlines restrictions that allow us to show this equivalence. We chose processor consistency since it is easier to implement and offers higher performance.

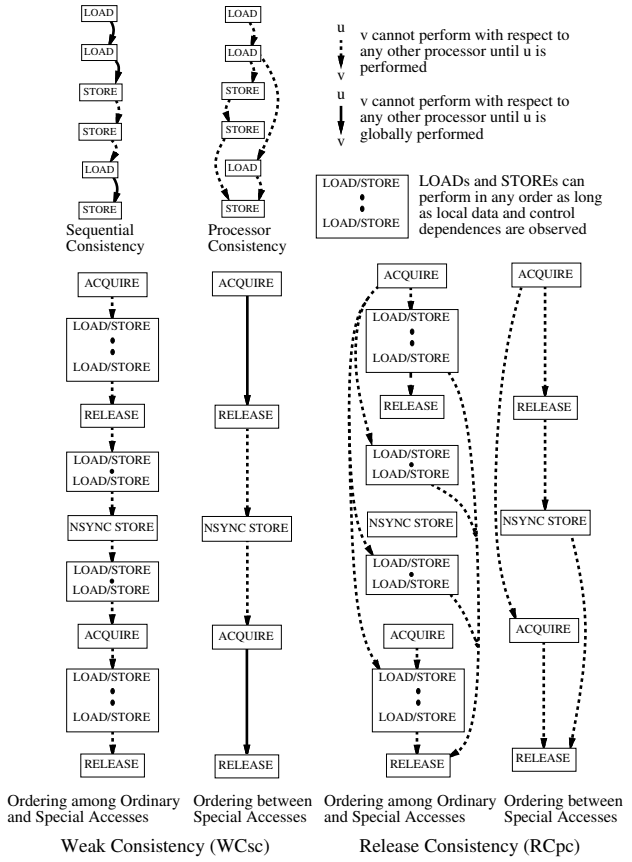

Figure 3: Ordering requirements for different consistency models.

4 Model Equivalences

The purpose of this section is to provide more insight into the similarities and differences among the consistency models presented in Sections 2 and 3 by showing relations and equivalences among the models.

We have presented four consistency models: sequential consistency (SC), processor consistency (PC), weak consistency with special accesses sequentially consistent (WCsc), and release consistency with special accesses processor consistent (RCpc). Two other models that fit within this framework are weak consistency with special accesses processor consistent (WCpc) and release consistency with special accesses sequentially consistent (RCsc). Figure 3 depicts the event orderings imposed by Conditions 2.1 through 2.3 for SC, PC, WCsc, and Condition 3.1 for RCpc. The WC and RC models have fewer restrictions on ordering than SC and PC, and RC has fewer restrictions than WC. Of course, a hardware implementation has the choice of enforcing the stated conditions directly or imposing some alternative set of conditions that guarantee the executions of programs appear as if the stated conditions were followed.

We define the relations $\geq$ (stricter) and $=$ (equal) for relating the models. If A and B are different consistency models, then relation $A \geq B$ says that results of executions of a program under model A will be in accordance to legal results for the program under model B, but not necessarily vice versa. The stricter relation is transitive. The relation $A = B$ says that for a certain program, models A and B cannot be distinguished based on the results of the program. Given $A \geq B$ and $B \geq A$, we know $A = B$. Some obvious relations that hold for any parallel program are: $SC \geq PC$, $SC \geq WCsc \geq RCsc$, $SC \geq WCpc \geq RCpc$, $PC \geq RCpc$, $WCsc \geq WCpc$, and $RCsc \geq RCpc$. However, the stricter relation does not hold among the following pairs: (PC,WCsc), (PC,RCsc), (PC,WCpc), and (RCsc,WCpc).

Due to the more complex semantics of the weaker models, it is desirable to show that the weaker models are equivalent to the stricter models for certain classes of programs. Such equivalences would be useful. For example, a programmer can write programs under the well defined semantics of the sequential consistency model, and as long as the program satisfies the restrictions, it can safely be ex-

ecuted under the more efficient release consistency model.

Let us first restrict the programs to PL programs under sequential consistency. Given such programs, we have proved the following equivalences: $SC = WCsc = RCsc$. This is done by proving $RCsc \geq SC$ for PL programs and using the relation $SC \geq WCsc \geq RCsc$. Our proof technique is based on an extension of the formalism presented by Shasha and Snir [13]. We have included the proof for $RCsc \geq SC$ in the appendix. A similar proof can be used to show $PC = WCpc = RCpc$ for PL programs under the processor consistency model.

More equivalences can be shown if we restrict programs to those that cannot distinguish between sequential consistency and processor consistency ($SC = PC$). Given a set of restrictions on competing LOAD accesses, it can be shown that $SC = PC$.[2] The restrictions are general enough to allow for all implementations of locks, semaphores, barriers, distributed loops, and task queues that we are interested in. Given competing LOAD accesses have been restricted (therefore, $SC = PC$) and shared accesses are properly labeled to qualify the program as a PL program under SC, it is easily shown that $SC = PC = WCsc = RCsc = WCpc = RCpc$. Therefore, such a program could be written based on the sequential consistency model and will run correctly under release consistency (RCpc).

The above equivalences hold for PL programs only. In some programs most accesses are competing (e.g., chaotic relaxation) and must be labeled as special for proper labeling. While this will make the equivalences hold, the program's performance may not be substantially better on RCsc than on SC. However, such applications are usually robust enough to tolerate a more relaxed ordering on competing accesses. For achieving higher performance in these cases, the programmer needs to directly deal with the more complex semantics of release consistency to reason about the program.

5 Performance Potentials for Different Models

The main purpose of examining weaker models is performance. In this section, we explore the potential gains in performance for each of the models. Realizing the full potential of a model will generally depend on the access behavior of the program and may require novel architectural and compiler techniques. Our goal is to provide intuition about how one model is more efficient than another.

The performance differences among the consistency models arise from the opportunity to overlap large latency memory accesses with independent computation and possibly other memory accesses. When the latency of an access is hidden by overlapping it with other computation, it is known as access *buffering*. When the latency of an access is hidden by overlapping with other accesses, it is known as access *pipelining*. To do buffering and pipelining for read accesses requires prefetch capability (non-blocking loads).

We provide simple bounds for the maximum performance gain of each model compared to a base execution model. The base model assumes that the processor is stalled on every access that results in a cache miss. It is easily shown that sequential consistency and processor consistency can at best gain a factor of 2 and 3, respectively, over the base model. This gain arises from the opportunity to buffer accesses. In practice though these two models are not expected to perform much better than the base model, since access buffering is not effective when the frequency of shared accesses is high.

The weak and release consistency models can potentially provide large gains over the base model, since accesses and computation in the region between two adjacent synchronization points can be overlapped freely as long as uniprocessor dependences are respected. In this case, the maximum gain over the base model is approximately equal to t_{lat}/t_{ser}, where t_{lat} is the latency of a miss and t_{ser} is the shortest delay between the issue of two consecutive accesses that miss in a cache. Intuitively, this is because ordinary accesses within a region can be pipelined. Unlike the maximum gains for SC and PC, the potential gains for WC and RC are more realizable. For example, several numerical applications fetch and update large arrays as part of their computations. The pipelining of reads and writes in such applications can lead to large performance gains.

The difference in performance between WC and RC arises when the occurrence of special accesses is more frequent. While weak consistency requires ordinary accesses to perform in the region between two synchronization points, release consistency relaxes this by allowing an ordinary access to occur anywhere between the previous acquire and the next release. In addition, an acquire can perform without waiting for previous ordinary accesses and ordinary accesses can perform without waiting for a release. Figure 4 shows an example that highlights the difference between the two models (assume that there are no local dependences).

To illustrate the performance gains made possible by the release consistency model, we consider the example of doing updates to a distributed hash table. Each bucket in the table is protected by a lock. A processor acquires the lock for a bucket first. Next, several words are read from records in that bucket, some computation is performed, and several words are written based on the result of the computation. Finally, the lock is released. The processor then moves on to do the same sequence of operations on another bucket. Such operations are common in several applications (for example, token hash tables in OPS5 [6]). The locality of data in such an application is low since the hash table can be large and several other processors may have modified an entry from the last time it was accessed. Therefore, the read and write accesses will miss often.

[2] Given such restrictions, one can allow an atomic test-and-set used as an acquire to perform before a previous special write access (e.g., unset) has been performed. We are currently preparing a technical report that describes the details.

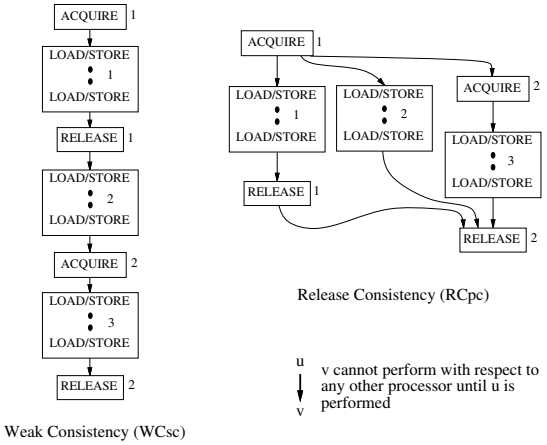

Figure 4: Possible overlap difference between WCsc and RCpc.

Under sequential consistency, all accesses and computation become serialized. With weak consistency, the reads can be pipelined. Of course, this assumes the architecture allows multiple outstanding reads. All reads need to complete before the computation. Once the computation completes, the writes occur in a pipelined fashion. However, before releasing the lock, all writes need to complete. The lock for the next record can not be acquired until the previous lock is released.

Release consistency provides the most opportunity for overlap. Within a critical section, the overlap is the same as in weak consistency. However, while the release is being delayed for the writes to complete, the processor is free to move on to the next record to acquire the lock and start the reads. Thus, there is overlap between the writes of one critical section and the reads of the next section.

To make the example more concrete, assume the latency of a miss is 40 cycles. Consider read miss, write miss, acquiring a lock, and releasing a lock as misses. Assume t_{ser} is 10 cycles and the computation time is 100 cycles. Assume three read misses and three write misses in each record lookup and update. If all accesses are serialized, each critical section takes 420 cycles. With weak consistency, the read misses before the computation and the write misses after the computation can be pipelined. The three read misses will complete in 60 cycles. The same is true for the write misses. Therefore, the critical section completes in 300 cycles on an implementation with weak consistency. Under release consistency, the same overlap is possible within a critical section. In addition, there is overlap between critical sections. Therefore, the processor can move on to the next critical section every 230 cycles. Figure 5 shows the overlap differences among sequential, weak, and release consistency. The segments shown span the time from the issue to the completion of an access. An access may be initiated by the processor several cycles before it is issued to the memory system.

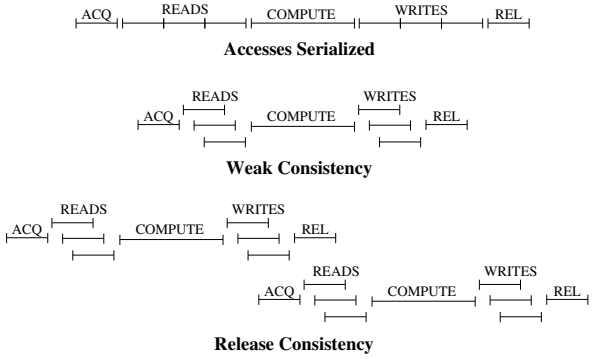

Figure 5: Overlap in processing hash table buckets.

6 Implementation Issues

The two most important issues from an implementation point of view are correctness and performance. The consistency model determines what a correct implementation of the memory system must provide. The challenge for a correct implementation is to achieve the full performance potential of the chosen consistency model. This section presents practical implementation techniques, focusing on issues relevant to scalable architectures that use caches, distributed memory, and scalable interconnection networks.

In the following subsections, we outline the techniques for ordering accesses under the various consistency models. The problem is split between ordering accesses to the same memory block and those to different memory blocks. General solutions to achieve the proper ordering are given along with the particular solutions employed in the DASH prototype system [8]. Our discussion focuses on invalidation-based coherence protocols, although the concepts can also be applied to update-based protocols.

6.1 Inter-Block Access Ordering and the FENCE Mechanism

As a result of the distribution of the memory and the use of scalable interconnection networks, requests issued by a processor to distinct memory modules may execute out of order. To maintain order among two accesses, we need a mechanism to delay the issue of one access until the previous one has been performed.[3] This requires each processor to keep track of its outstanding accesses. Due to multiple paths and variable delays within the memory system, acknowledge messages from target memories and caches are required to signal the completion of an access.

[3]There is a subtle difference between delaying issue and delaying an access from being performed with respect to any other processor. Instead of delaying the issue of a write, the processor can delay making the new value visible to other processors. The write is considered performed when the new value is made visible to other processors. This allows write accesses to be pipelined. We are studying hardware techniques that exploit this distinction for write accesses in invalidate-based machines. However, we do not consider such techniques in this paper.

Model	*Operation Preceded by Fence*	*Fence Type*	*Previous Accesses that must be performed*	
			LOAD	STORE
SC	LOAD	full	G	P
	STORE	full	G	P
PC	LOAD	full	P	
	STORE	write	P	P

Figure 6: Fence operations to achieve sequential and processor consistency. P denotes performed while G denotes globally performed.

We refer to the mechanism for delaying the issue of accesses as a *fence* [3, 5, 13]. We define a general set of fence operations and demonstrate how these fence operations can be used to implement the consistency models presented earlier. While fence operations are described here as explicit operations, it is possible, and often desirable, to implicitly associate fences with load, store, and special (e.g., acquire, release) accesses.

For generality, we assume that load operations are non-blocking. The processor can proceed after the load is issued, and is only delayed if the destination register of the load is accessed before the value has returned. In contrast, a blocking load stalls the processor until it is performed.

Fence operations can be classified by the operations they delay and the operations they wait upon. Useful operations to delay are: (i) all future read and write accesses (*full fence*); (ii) all future write accesses (*write fence*), and (iii) only the access immediately following the fence (*immediate fence*). Likewise, useful events to wait for are a combination of previous load accesses, store accesses, and (for the weaker models) special accesses.

Figure 6 shows the placement and type of fence operations required to achieve sequential and processor consistency. For example, the first line for SC in the figure indicates that the fence prior to a load is a full fence waiting for all previous loads to globally perform and all previous stores to perform. Figure 7 shows the fence operations necessary to achieve weak consistency (WCsc) and release consistency (RCpc). The implementations outlined are the most aggressive implementation for each model in that only the delays that are necessary are enforced. Conservative implementations are possible whereby hardware complexity is reduced by allowing some extra delays.

To implement fences, a processor must keep track of outstanding accesses by keeping appropriate counters. A count is incremented upon the issue of the access, and is decremented when the acknowledges come back for that access (an acknowledge for a read access is simply the return value). For full and write fences, the number of counters necessary is a function of the number of different kinds of accesses that need to be distinguished. For example, RCpc needs to distinguish four groups of accesses: ordinary, nsync load, acquire, and special store accesses. Therefore, an aggressive implementation requires four counters. However, only two counters are required if special loads are blocking. For immediate fences, the same number of counters (as for full or write fence) is required for each outstanding immediate fence. Therefore, we have to multiply this number by the number of immediate fences that are allowed to be outstanding. Slightly conservative implementations of release consistency may simply distinguish special load accesses from other accesses by using two counters (only one if special loads are blocking) and limit the number of outstanding immediate fences to a small number.

Full fences can be implemented by stalling the processor until the appropriate counts are zero. A write fence can be implemented by stalling the write buffer. The immediate fence, which is only required in release consistency (for an aggressive implementation), requires the most hardware. Each delayed operation requires an entry with its own set of counters. In addition, accesses and acknowledges need to be tagged to distinguish which entry's counters should be decremented upon completion. In the DASH prototype (discussed in Section 6.3), a write fence is substituted for the immediate fence (load accesses are blocking), thus providing a conservative implementation of release consistency.

6.2 Intra-Block Ordering of Accesses

The previous section discussed ordering constraints on accesses to different memory blocks. When caching is added to a multiprocessor, ordering among accesses to the same block becomes an issue also. For example, it is possible to receive a read request to a memory block that has invalidations pending due to a previous write. There are subtle issues involved with servicing the read request while invalidations are pending. Cache blocks of larger than one word further complicate ordering, since accesses to different words in the block can cause a similar interaction.

In an invalidation-based coherence protocol, a store operation to a non-dirty location requires obtaining exclusive ownership and invalidating other cached copies of the block. Such invalidations may reach different processors at different times and acknowledge messages are needed to indicate that the store is performed. In addition, ownership accesses to the same block must be serialized to ensure only one value persists. Unfortunately, the above two measures are not enough to guarantee correctness. It is important to distinguish between dirty cache lines with pending invalidates versus those with no pending invalidates. Otherwise, a processor cache may give up its ownership to a dirty line with invalidates pending to a read or write request by another processor, and the requesting processor would not be able to detect that the line returned was not performed. The requesting processor could then improperly pass through a fence operation that requires all previous loads to be globally performed (if access was a read) or all previous stores to be performed (if access was a write). Consequently, read and ownership requests to a

Model	*Operation Preceded by Fence*	*Fence Type*	*Previous Accesses that must be Performed*			
			LOAD	STORE	SPECIAL LD	SPECIAL ST
WCsc	first LOAD/STORE after SPECIAL	full			P	P
	SPECIAL LD	full	P	P	G	P
	SPECIAL ST	full	P	P	G	P

Model	*Operation Preceded by Fence*	*Fence Type*	*Previous Accesses that must be Performed*					
			LOAD	STORE	NSYNC LD	ACQUIRE	NSYNC ST	RELEASE
RCpc	first LOAD/STORE after ACQUIRE	full				P		
	NSYNC LD	immediate			P	P		
	ACQUIRE	full			P	P		
	NSYNC ST	immediate			P	P	P	P
	RELEASE	immediate	P	P	P	P	P	P

Figure 7: Fence operations to achieve weak consistency and release consistency. P denotes performed while G denotes globally performed.

block with pending invalidates must either be delayed (by forcing retry or delaying in a buffer) until the invalidations are complete, or if the request is serviced, the requesting processor must be notified of the outstanding status and acknowledges should be forwarded to it to indicate the completion of the store. The first alternative provides atomic store operations. The second alternative doesn't guarantee atomicity of the store, but informs the requesting processor when the store has performed with respect to all processors. In the next subsection, we will discuss the specific implementation technique used in DASH.

The issues in update-based cache coherence schemes are slightly different. In an update-based scheme, a store operation to a location requires updating other cache copies. To maintain coherence, updates to the same block need to be serialized at a central point and updates must reach each cache in that order. In addition, SC-based models are difficult to implement because copies of a location get updated at different times (it is virtually impossible to provide atomic stores). Consequently, a load may return a value from a processor's cache, with no indication of whether the responsible store has performed with respect to all processors. For this reason, PC-based models are an attractive alternative for update-based coherence schemes.

6.3 The DASH Prototype

The DASH multiprocessor [8], currently being built at Stanford, implements many of the features discussed in the previous sections. The architecture consists of several processing nodes connected through a low-latency scalable interconnection network. Physical memory is distributed among the nodes. Each processing node, or *cluster*, is a Silicon Graphics POWER Station 4D/240 [2] consisting of four high-performance processors with their individual caches and a portion of the shared memory. A bus-based snoopy scheme keeps caches coherent within a cluster while inter-cluster coherence is maintained using a distributed directory-based protocol. For each memory block, the directory keeps track of remote clusters caching it, and point-to-point messages are sent to invalidate remote copies of the block.

Each cluster contains a directory controller board. This directory controller is responsible for maintaining cache coherence across the clusters and serving as the interface to the interconnection network. Of particular interest to this paper are the protocol and hardware features that are aimed at implementing the release consistency model. Further details on the protocol are given in [8].

The processor boards of the 4D/240 are designed to work only with the simple snoopy protocol of the bus. The base, single-bus system implements a processor consistency model. The single bus guarantees that operations cannot be observed out of order, and no acknowledgements are necessary. Read operations are blocking on the base machine.

In the distributed DASH environment, the release consistency model allows the processor to retire a write after it has received ownership, but before the access is performed with respect to all other processors. Therefore, a mechanism is needed to keep track of outstanding accesses. In DASH, this function is performed by the remote access cache (RAC). Corresponding to each outstanding access, the RAC maintains a count of invalidation acknowledges pending for that cache block and keeps track of the processor(s) associated with that access. In addition, the RAC maintains a counter per processor indicating the number of RAC entries (i.e., outstanding requests) in use by each processor.

To ensure proper intra-block ordering, the RAC detects accesses to blocks with pending invalidates by snooping on the cluster bus. In case of a local processor access, the

RAC allows the operation to complete, but adds the new processor to the processor tag field of the RAC. Thus, the processor that has a copy of the line now shares responsibility for the block becoming performed. For remote requests (i.e., requests from processors on a different cluster) the RAC rejects the request. The RAC does not attempt to share a non-performed block with a remote processor because of the overhead of maintaining the pointer to this remote processor and the need to send an acknowledgement to this processor when the block has been performed. Rejecting the request is not as desirable as queuing the requests locally, but this would require extra buffering.

To ensure proper inter-block ordering, DASH again relies on the acknowledges in the protocol and the RAC. The per processor counter indicates the number of outstanding requests for each processor. When this count is zero, then the processor has no outstanding operations and a fence operation can complete. There are two types of fence operations in DASH: a full fence and a write fence. The full fence is implemented by stalling the processor until all previous memory operations are performed (i.e., the RAC count is zero for that processor). The less restrictive write fence is implemented by stalling the output of the processor's write-buffer until all previous memory operations are performed. This effectively blocks the processor's access to the second level cache and cluster bus.

DASH distinguishes lock and unlock synchronization operations by physical address. All synchronization variables must be partitioned to a separate area of the address space. Each unlock (release) operation includes an implicit write fence. This blocks the issuing of any further writes (including the unlock operation) from that processor until all previous writes have been performed. This implicit write fence provides a sufficient implementation for release consistency. The explicit forms of full and write fence operations are also available. These allow the programmer or compiler to synthesize other consistency models.

7 Concluding Remarks

The issue of what memory consistency model to implement in hardware is of fundamental importance to the design of scalable multiprocessors. In this paper, we have proposed a new model of consistency, called release consistency. Release consistency exploits information about the property of shared-memory accesses to impose fewer restrictions on event ordering than previously proposed models, and thus offers the potential for higher performance. To avoid having the programmer deal directly with the more complex semantics associated with the release consistency model, we presented a framework for distinguishing accesses in programs so that the same results are obtained under RC and SC models. In particular, we introduced the notion of properly-labeled (PL) programs and proved the equivalence between the SC and the RCsc model for PL programs. This is an important result since programmers can use the well defined semantics of sequential consistency to write their programs, and as long as the programs are PL, they can be safely executed on hardware implementing the release consistency model.

To implement the various consistency models, we propose the use of fence operations. Three different kinds of fence operations – full fence, write fence, and immediate fence – were identified. Careful placement of these multiple types of fences enabled us to minimize the duration for which the processor is blocked. We also discussed subtle ordering problems that arise in multiprocessors with caches and provided solutions to them. Finally, practical implementation techniques were presented in the context of the Stanford DASH multiprocessor.

We are currently building the prototype for the DASH architecture, which supports the release consistency model. We are using a simulator for the system to quantify the performance differences among the models on real applications and to explore alternative implementations for each model. We are also exploring compiler techniques to exploit the less strict restrictions of release consistency. Finally, we are investigating programming language and programming environment enhancements that allow the compiler to gather higher level information about the shared accesses.

8 Acknowledgments

We would like to thank Rohit Chandra for several useful discussions, and Jaswinder Pal Singh and Sarita Adve for their comments on the paper. We also wish to thank the reviewers for their helpful comments. This research was supported by DARPA contract N00014-87-K-0828. Daniel Lenoski is supported by Tandem Computer Incorporated. Phillip Gibbons is supported in part by NSF grant CCR-86-10181 and DARPA contract N00014-88-K-0166.

References

[1] Sarita Adve and Mark Hill. Personal communication. March 1990.

[2] Forest Baskett, Tom Jermoluk, and Doug Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proceedings of the 33rd IEEE Computer Society International Conference – COMPCON 88*, pages 468–471, February 1988.

[3] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 782–789, 1985.

[4] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors.

In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[5] James R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SCI Committee, March 1989.

[6] Anoop Gupta, Milind Tambe, Dirk Kalp, Charles Forgy, and Allen Newell. Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2):95–124, 1988.

[7] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[8] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[9] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

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

[11] Christoph Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989.

[12] G. E. Schmidt. The Butterfly parallel processor. In *Proceedings of the Second International Conference on Supercomputing*, pages 362–365, 1987.

[13] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

Appendix A: Proof for SC = RCsc

In this appendix we present a proof of the equivalence between SC and $RCsc$ for PL programs (with respect to SC). For brevity, we will use the terms RC to denote $RCsc$ and PL to denote PL programs properly-labeled with respect to SC. We begin with a few definitions.

An *execution* of a program on an implementation defines a pair, (T, EO), as follows.

- The *per-processor trace*, T, is a set of traces, one for each processor, showing the instructions executed by the processor during the execution. The order among instructions in the trace is adjusted to depict program order for each processor.
- The execution order, EO, specifies the order in which conflicting accesses are executed. (Recall from section 3 that two accesses, u and v, *conflict* if and only if u and v are to the same location and one is a STORE.) EO fully specifies the results of a program, since any sequential execution of the accesses in an order that extends the execution order (i.e., topological sort) will give the same result.

The *delay relation*, D, is an ordering constraint among instructions within a processor as imposed by some event ordering. For example, the delay relation for RC enforces Condition 3.1, as well as local data and control dependences. These notions of execution order, conflicting accesses, and delay relation were developed previously in [13]. To prove various equivalences, we extend the notions presented in [13] to handle conditionals, non-atomic writes, and consistency models other than SC (we are preparing a technical report on this). Although writes are not atomic, we can assume that conflicting accesses are totally ordered by EO since the implementations we are considering provide cache coherence (i.e., all processors observe two writes to the same location in the same order). Also we make the common assumption that accesses are only to words of memory: each read access returns the value written by some (single) write access.

The execution order EO on an implementation is considered legal if $EO \cup D$ is acyclic. The graph corresponding to $EO \cup D$ is called the *precedence graph*, G, of the execution. Thus a cycle in G denotes an impossible execution. An instruction x *reaches* an instruction y in an execution if there is a (directed) path from x to y in the precedence graph of the execution.

We partition EO into two disjoint sets, EO_s and EO_o, where EO_s defines the execution order among any two (conflicting) special accesses and E_o defines the execution order among any two (conflicting) accesses where at least one is an ordinary access. Likewise, G is partitioned into G_s and G_o.

Given these preliminary definitions, we now proceed with the proof. We first assume that special accesses are not affected by ordinary accesses. This permits us to claim that $EO_{s:SC} = EO_{s:RC}$ follows if $T_{SC} = T_{RC}$. We will later describe how this restriction can be lifted. In lemma 1, we show that if the same per-processor trace can occur on both SC and RC, then the program results are the same. This lemma is then used to prove the main theorem, which shows that $SC = RC$ for all PL programs. The difficulty in extending the lemma to the main theorem is in showing that any legal trace on RC may occur on SC despite any conditional branches or indirect addressing. Note that $SC \geq RC$ for any program, so it suffices to show that $RC \geq SC$.

Lemma 1: Consider an execution $E = (T_{RC}, EO_{RC})$ on RC of a PL program. If there exists a trace on SC such that $T_{SC} = T_{RC}$, then there is a corresponding execution on SC with the same results (i.e., $EO_{SC} = EO_{RC}$).

Proof: Since the event ordering on special accesses is SC for both implementations, and special accesses are not affected by ordinary accesses, $G_{s:SC} = G_{s:RC}$ is a legal precedence graph for special accesses on SC. We will show there exists a legal execution on SC, based on $G_{s:SC}$, such that $EO_{o:SC} = EO_{o:RC}$.

Let u and v be two conflicting accesses from T_{SC}, such that u is an ordinary access. If u and v are on the same processor, then the execution order, EO, between the two is determined by local dependences and is enforced in the same way on SC and RC.

If u and v are on different processors, then the two accesses need to be ordered through special accesses for the program to be a PL program. Access v can be either an ordinary or a special access. Consider the case where v is an ordinary access. For u and v to be ordered, there is either (a) a release REL_u and an acquire ACQ_v such that REL_u reaches ACQ_v in $G_{s:SC}$ or (b) a release REL_v and an acquire ACQ_u such that REL_v reaches ACQ_u in $G_{s:SC}$. If (a) holds, then u before v, $uEOv$, is the only possible execution order on SC. The same is true on RC, since $vEOu$ will lead to a cycle in the precedence graph. This is because clauses (A) and (B) of Condition 3.1 are upheld. Likewise, a symmetric argument can be used if (b) holds. The same correspondence between SC and RC can be shown for the case where v is a special access. Thus the execution order EO between u and v is the same on SC and RC.

Since $EO_{s:SC} = EO_{s:RC}$, and this execution order determines an E_o that is the same for both SC and RC, we have shown that $EO_{SC} = EO_{RC}$. □

Therefore, $RC \geq SC$ for a program if, for every execution of a program on RC, there is an execution on SC such that the traces are the same.

How can the traces for a program on SC and RC differ? There are two possible sources for any discrepancies between traces: conditional control flow (affecting which instructions are executed) and indirect addressing (affecting the location accessed by a read or write instruction). In what follows, we consider only conditionals. Extending the argument to handle programs with indirect addressing is trivial, and omitted in this proof.

We will prove that $SC = RC$ for PL programs as follows. We must show that there exists an execution on SC in which the outcome of each conditional is the same. A conditional for which we have shown this correspondence will be designated *proven*, otherwise it will be called *unproven*. Initially, all conditionals in the trace on RC are *unproven*. We will construct the trace on SC inductively in a series of stages, where at each stage, we show that an unproven conditional occurs the same way on SC. Once all conditionals are proven, the traces must be equal and we can apply lemma 1.

Theorem 2: $SC = RC$ for PL programs.

Proof: Let P be a PL program. Consider any execution $E = (T_{RC}, EO_{RC})$ on RC. Let G_{RC} be the precedence graph for E. By the definition of a precedence graph, any instruction that affected another instruction in E, e.g., affected the existence of a write access or the value returned on a read access, reaches that instruction in G_{RC}.

As indicated above, we proceed in a series of stages, one for each conditional. At each stage, we construct an execution on SC such that some unproven conditional and all previously proven conditionals have the same outcome on SC and RC.

We begin with stage 1. The proof for stage 1 will be shown using a series of claims. As we shall see, the proof for each remaining stage is identical to stage 1.

Since G_{RC} is acyclic, there is at least one unproven conditional, u_1, that is not reached by any other unproven conditional. Let p_{u_1} be the processor that issued u_1. Let A_1 be the set of instructions that reach u_1 in G_{RC}. Although A_1 is only a subtrace (not even a prefix) of the entire execution E, we will show that the set A_1, constructed in this way, can be used to prove u_1.

Let A_{1s} be the special accesses in A_1. We have the following characterization of A_{1s}.

Claim 1: All special accesses program ordered prior to an access in A_{1s} are themselves in A_{1s}. There are no special accesses within any branch of an unproven conditional, u, where u is program ordered prior to an access in A_{1s}.

Proof: We first show that the claim holds for acquires. Any acquire program ordered prior to an access, x, in A_1 reaches x and hence will itself be in A_{1s}. There are no acquires within any branch of an unproven conditional program ordered prior to an access in A_{1s} since no access after such a conditional can complete prior to the conditional itself.

We claim that the last program ordered access in A_1 for each processor (other than p_{u_1}) is a special access. This fact can be shown by contradiction. Let z_1, an ordinary access, be the last program ordered access for some processor in A_1 (other than p_{u_1}). Since z_1 is in A_1, there is a path, $z_1, z_2, \ldots, u_1$, in G_{RC}. No access in A_1 is locally dependent on z_1 since it is the last program ordered access on its processor. Since P is a PL program, a release below z_1 is needed to order the access ahead of z_2 on SC. However, there is no release below z_1 in A_1. Thus the only way for z_1 to affect z_2 on RC would be in a competing manner that was prevented on SC. This can happen only if some acquire above either z_1 or z_2 were missing in A_{1s}, which contradicts the claim of the previous paragraph.

Claim 1 follows since program order is preserved on RC for special accesses. □

Given this characterization of A_{1s}, we show that there is an execution on SC such that special accesses are the same as in A_1. In other words, we show that both implementations have the same G_s for A_1. This will be used to show that the results returned by read accesses are the same and hence the outcome of conditional u_1 is the same.

Claim 2: There is a prefix of an execution on SC such that the special accesses are precisely the accesses in A_{1s} and the execution order among these special accesses is identical to $EO_{s:RC}$.

Proof: The special accesses in A_{1s} are self-contained, i.e., there are no acquires in A_{1s} that are waiting on releases not in A_{1s}. By claim 1, there is an execution on SC such that all special accesses in A_{1s} occur. Since special accesses are SC on both implementations, the same execution order among these special accesses is possible on both. To complete the proof, we argue that no other special access (i.e., not in $A1_s$) can be forced to occur prior to an access in $A1_s$ in every execution on SC that includes A_{1s}. How can a special access be forced to occur on SC? Either the special access is program ordered prior to some access in A_{1s} or it is a release satisfying an acquire that is not satisfied in A_{1s}. But the former case contradicts claim 1 and the latter case contradicts A_{1s} being self-contained. Thus there is an execution on SC and a point in this execution in which the special accesses performed are precisely the accesses in $A1_s$, and the execution order among these special accesses is identical to $EO_{s:RC}$. □

Claim 3: There is an execution on SC in which the outcome of u_1 is the same as in E.

Proof: Since A_1 consists of all instructions that affect u_1 in E, the outcome of u_1 in the full execution E is determined by only the accesses in A_1. Thus it suffices to show that (a) there is an execution E_{SC} on SC in which the instructions in A_1 occur, (b) all read accesses in A_1 return the same results in E_{SC} as in E, and (c) the outcome of u_1 in E_{SC} is determined by only the accesses in A_1.

The accesses in A_1 will occur on SC since none of them are within an unproven conditional. This follows from the fact that if an access within a conditional can reach u_1, then so can its conditional (since RC enforces control dependence).

Consider the prefix execution, E_1, constructed in claim 2, and let EO_{1s} be the execution order among special accesses in A_1. Since E_1 is a prefix of a PL program, EO_{1s} determines $EO_{o:SC}$ for the accesses in A_1.

We claim that EO_{1s} determines $EO_{o:RC}$ for the accesses in A_1. We must show that the instructions in E_1 that are not in A_1 have no effect on the results returned by read accesses in A_1. Consider a write access, w_1, in E_1 that reaches a read access, r_1, in A_1 on SC, but does not reach it in G_{RC}. Since r_1 is in A_1, it cannot be reached on G_{RC} by an unproven conditional. Thus any local dependence chain from w_1 to r_1, inclusive, does not include any instruction within an unproven conditional. Hence, if there is a local dependence on SC, then there will be one on RC. Moreover, if w_1 is ordinary, then it must be followed by a release on SC. Since all accesses complete on RC prior to a release, w_1 must be in A_1 and reach the release in G_{RC}. Since EO_{1s} is the execution order for both SC and RC, w_1 must reach r_1 in G_{RC}. Similarly, if w_1 is a special access, it must reach r_1 in G_{RC}. In either case, we have a contradiction.

Therefore, the results returned by read accesses in A_1 on SC depend only on other accesses in A_1. Thus we can view the traces as being the same. Hence by lemma 1, all read accesses in A_1 up to the last special access on p_{u_1} return the same results in E_{SC} as in E.

Finally, the outcome of conditional u_1 depends on the values read by p_{u_1}. These read accesses can be ordinary or special. Since P is a PL program, an ordinary read access affecting u_1 returns the value of a write access, w_1, that is ordered by local dependence or through an acquire. A special read access affecting u_1 is already shown to return the correct value. Thus the outcome of u_1 is the same as in E. □

Stage $k > 1$. Inductively, we can assume that $k-1$ unproven conditionals have been shown to correspond on SC and RC, such that there is a k^{th} unproven conditional, u_k, that is not reached by any other unproven conditional. At this stage, we add to the current subtrace all instructions that can reach u_k. Let A_k be this new set of instructions. As before, although A_k is not a complete trace on SC (or even a prefix), we can argue that there is at least one execution on SC such that (1) the same G_s occurs on A_k in both SC and RC, and thus (2) the outcome of u_k is the same as in E. The arguments are identical to those in claims 1–3 above, where $u_1, \ldots, u_{k-1}$ are no longer unproven conditionals.

Therefore, by induction, there is an execution on SC such that the outcome of all conditionals is the same as in E. Since all unprovens correspond, we know that the full traces are equal. Thus there exists a valid trace T_{SC} of P on SC such that $T_{SC} = T_{RC}$. Hence by lemma 1, there exists an execution on SC such that $E_{SC} = E_{RC}$, i.e., the results are the same. This shows that $RC \geq SC$ for P. Since $SC \geq RC$, it follows that $RC = SC$ for P. □

We have assumed for the above proof that special accesses are not affected by ordinary accesses. This is used in the proof, for example, when we assume in lemma 1 that $EO_{s:SC} = EO_{s:RC}$ follows if $T_{SC} = T_{RC}$. In general, however, an ordinary access can affect a special access, e.g., it can be to the same location. Our proof can be extended to handle this general case in which special accesses are affected by ordinary accesses, as follows. Consider special read accesses, conditional branches, and accesses with indirect addressing all to be initially unproven. As above, include one new unproven at each stage, until all are proven. Since we are proving special read accesses along the way, we ensure the correspondence among special accesses between SC and RC at each stage (i.e., $EO_{s:SC} = EO_{s:RC}$). Therefore, theorem 2 holds for general PL programs.

Adve and Hill [1] have proved a similar equivalence between sequential consistency and their version of weak ordering.