Specifying System Requirements for Memory Consistency Models

Kourosh Gharachorloo
Sarita V. Adve
Anoop Gupta
John L. Hennessy
Mark D. Hill

Technical Report #1199

December 1993

Specifying System Requirements for Memory Consistency Models *

Kourosh Gharachorloo[†], Sarita V. Adve[‡],
Anoop Gupta[†], John L. Hennessy[†], and Mark D. Hill[‡]

[†]Computer System Laboratory
Stanford University
Stanford, CA 94305

[‡]Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

Stanford University Technical Report CSL-TR-93-594
University of Wisconsin-Madison Computer Sciences Technical Report #1199

Abstract

The memory consistency model of a shared-memory system determines the order in which memory accesses can be executed by the system, and greatly affects the implementation and performance of the system. To aid system designers, memory models either directly specify, or are accompanied by, a set of low-level system conditions that can be easily translated into a correct implementation. These sufficient conditions play a key role in helping the designer determine the architecture and compiler optimizations that may be safely exploited under a specific model. Therefore, these conditions should obey three important properties. First, they should be unambiguous. Second, they should be feasibly aggressive; i.e., they should not prohibit practical optimizations that do not violate the semantics of the model. Third, it should be relatively straightforward to convert the conditions into efficient implementations, and conversely, to verify if an implementation obeys the conditions. Most previous approaches in specifying system requirements for a model are lacking in at least one of the above aspects.

This paper presents a methodology for specifying the system conditions for a memory model that satisfies the above goals. A key attribute of our methodology is the exclusion of ordering constraints among memory operations to different locations by observing that such constraints are unnecessary for maintaining the semantics of a model. To demonstrate the flexibility of our approach, we specify the conditions for several proposed memory models within this framework. Compared to the original specification for each model, the new specification allows more optimizations without violating the original semantics and, in many cases, is more precise.

1 Introduction

A *memory consistency model* or *memory model* for a shared-memory multiprocessor specifies how memory operations of a program will appear to execute to the programmer. The most commonly assumed model is *sequential*

*The Stanford University authors are supported by DARPA contract N00039-91-C-0138. Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award. The University of Wisconsin authors are supported in part by a National Science Foundation Presidential Young Investigator Award (MIPS-8957278) with matching funds from A.T. & T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation. Sarita Adve is also supported by an IBM graduate fellowship.

[illegible]

consistency (SC) [Lam79]. While SC provides an intuitive model for the programmer, it restricts many architecture and compiler optimizations that exploit the reordering and overlap of memory accesses [DSB86, MPC89]. This has led researchers to propose alternate models that allow more optimizations: e.g., *processor consistency* (PC) [GLL+90], *total store ordering* (TSO) [SUN91], *partial store ordering* (PSO) [SUN91], *weak ordering* (WO) [DSB86], and *release consistency* (RCsc/RCpc) [GLL+90].[1] These models are referred to as *hardware-centric* because they are defined in terms of relatively low-level hardware constraints on the ordering of memory accesses.

While the above relaxed models provide substantial performance improvements over SC [GGH91, GGH92, ZB92], they are difficult for programmers to reason with. To remedy this, another category of models, referred to as *programmer-centric*, have been proposed. Programmer-centric models provide a higher level system abstraction to the programmer, thus relieving the programmer from directly reasoning with memory access optimizations. Instead, the programmer can reason with SC and is only required to provide certain information about memory accesses (e.g., which accesses may be involved in a race [AH90b, GLL+90]). This information is in turn used to allow optimizations without violating SC. *DRF0* [AH90b], *DRF1* [AH93], *PL* [GLL+90], and *PLpc* [GAG+92] are example programmer-centric models that allow optimizations similar to the hardware-centric models.

To correctly and efficiently implement a memory model, the system designer must identify the hardware and software optimizations that are allowed by that model. Hardware-centric models, by the nature of their specification, directly specify such optimizations. In contrast, it is difficult to deduce such optimizations from the specification of programmer-centric models. Consequently, to aid designers, these models are also typically accompanied by a set of low-level system conditions that are proven sufficient for correctness.

For both classes of models, the low-level system specification plays a key role by directly influencing the hardware and system software implementation. Therefore, it is important for the specification to satisfy the following criteria. First, the specification should be precise and complete. Any ambiguity arising from the specification is undesirable and can lead to incorrect implementations. Second, the specification should be general, allowing a wide range of system designs and optimizations. To achieve this goal, the specification should impose as few constraints as necessary to maintain the semantics of the model. Finally, it should be easy to identify the allowed optimizations from the specification and conversely to determine if a particular implementation obeys the specification. For many of the models described above, the specifications of system constraints as they appear in the literature fall short of meeting these goals by being overly restrictive or ambiguous.

This paper presents a framework for specifying sufficient system constraints for a memory model that meets the above criteria. Our framework extends previous work by Collier [Col92], Sindhu et al. [SFC91], and Adve and Hill [AH92]. The framework consists of two parts: an abstraction of a shared-memory system, and a specification methodology for system requirements based on that abstraction. Our abstraction extends previous work by adequately modeling essential characteristics of a shared-memory system, such as replication of data, non-atomicity of memory operations, and allowing a processor to read the value of its own write before the write takes place in any memory copies. Previous abstractions either do not directly model some of these characteristics or are more complex. A key attribute of our specification methodology is the exclusion of ordering constraints among operations to different locations. This allows us to expose more optimizations while still maintaining the original semantics of a model. To demonstrate the flexibility of our approach, we specify the system requirements for several of the models discussed above. In all cases, the new specifications expose more potential optimizations than the original specifications. In some cases, the new specifications are also more precise than the original specifications. Furthermore, expressing the specifications within a uniform framework allows for an accurate comparison of the system constraints imposed by the different models.

The rest of the paper is organized as follows. Section 2 presents our abstraction for the system and our framework, and describes how system requirements may be specified through several examples. Section 3 describes how the specifications can be translated into implementation constraints for the architecture and the compiler. Section 4 compares our abstraction with previously proposed abstractions and discusses some implications of our approach. Finally, we conclude in Section 5. The detailed specification for several models with the corresponding correctness proofs are provided in Appendices B and C; Appendices A and D provide some of the other conditions that are required for correctness.

[1]The processor consistency model considered in this paper is different from that proposed by Goodman [Goo91]. The definitions for PC and RCsc/RCpc given in [GLL+90] are modified in a minor way as explained in [GGH93].

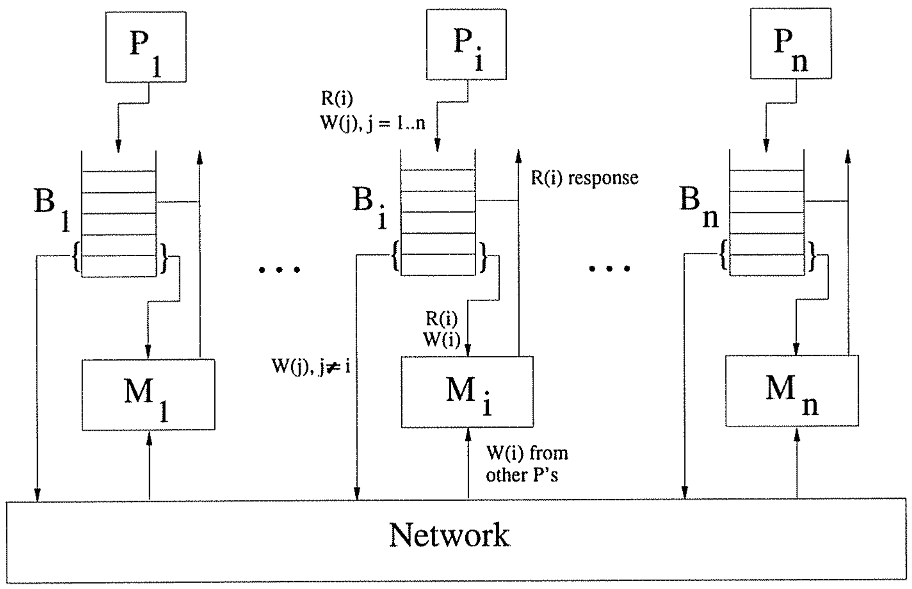


Figure 1: Abstraction for a shared-memory system.

2 Our Framework

This section describes our framework for specifying system requirements that are imposed by a memory model. The first part presents an abstraction of a shared-memory system that serves as the basis for the framework. We then formally define the notion of an execution and describe how conditions may be imposed on an execution in order to satisfy the semantics of various memory models. Our abstraction is influenced by the abstractions of Collier [Col92] and that of Sindhu et al. [SFC91, SUN91]. Some of the formalism and terminology are derived from work by Adve and Hill [AH92].

2.1 Abstraction of the System

Figure 1 shows the various components in our abstraction. The system consists of n processors, $P_1, \ldots, P_n$.[2] Each processor has a *complete* copy of the shared (writable) memory, denoted as M_i for P_i. A *read operation* reads a specific memory location and a *write operation* modifies a location. A read operation R by P_i is comprised of a single atomic sub-operation, *R(i)*. A write operation W by P_i is comprised of *(n+1)* atomic sub-operations: an initial write sub-operation $W_{init}(i)$ and n sub-operations *W(1), ..., W(n)*.[3] Each processing node also has a (conceptually infinite) memory buffer, denoted as B_i for P_i. A read operation on P_i results in the read sub-operation R(i) being placed into the corresponding buffer, B_i. Similarly, a write operation on P_i results in write sub-operations W(1), ..., W(n) being placed in B_i. The sub-operation W_{init}(i) of W corresponds to the initial event of placing W(1), ..., W(n) into B_i. Sub-operations are removed from B_i (not necessarily in first-in-first-out order) and then issued to the memory system. A write sub-operation W(j) by P_i executes when it is issued from B_i to the memory system and atomically updates its destination location in the memory copy M_j of P_j to the specified value. A read sub-operation R(i) by P_i executes when its corresponding read operation is issued to the memory system and returns a value. If there are any write sub-operations W(i) in B_i that are to the *same* location as R(i), then R(i) returns the value of the last such W(i) that was placed in the buffer (and is still in B_i). Otherwise, R(i) returns the value of the last write sub-operation W(i) that executed in the memory copy M_i of P_i.

The above abstraction is a conceptual model meant to only capture the important properties of shared-memory systems relevant to specifying system requirements for memory models. For example, we do not model details of the processor architecture because we assume that each processor by itself behaves like a correct uniprocessor (this

[2]These can be thought of as virtual processors. In this way, the notion of process or processor become interchangeable if there is one process per virtual processor.

[3]All n sub-operations may not be present in an execution; see Condition 5.

notion is discussed further in the next section). Similarly, we only model memory operations to shared writable data. Below, we explain how the different features of the above abstraction (i.e., a complete copy of memory for each processor, several atomic sub-operations for a write, and buffering operations before issue to memory) adequately model shared-memory systems.

The first two features of a complete memory copy per processor and several atomic sub-operations for writes are based directly on the abstraction by Collier [Col92] and have been previously used as the basis for specifying system requirements for memory models (e.g., DRF1 [AH92]). Dubois et al.'s "perform with respect to" abstraction [DSB86, SD87], though different in flavor, also effectively captures these concepts. Providing each processor with a copy of memory serves to model the multiple copies of a datum that are present in real systems due to the replication and caching of shared data. For example, in reality the copy of memory modeled for a processor may represent a union of the state of the processor's cache, and blocks belonging to memory or other caches that are not present in this processor's cache.

The multiple sub-operations attributed to each write operation model the fact that updating multiple copies of a location may be non-atomic. Adve and Hill [AH92] explain the correspondence to real systems as follows (slightly paraphrased). "While there may be no distinct physical entity in a real system that corresponds to a certain sub-operation, a logically distinct sub-operation may be associated with every operation and a memory copy. For example, updating the main memory on a write corresponds to the sub-operations of the write in memory copies of processors that do not have the block in their cache. Also, while sub-operations may not actually execute atomically in real systems, one can identify a single instant in time at which the sub-operation takes effect such that other sub-operations *appear* to take effect either before or after this time." Note that in reality, write operations may actually invalidate a processor's copy instead of updating it with new data. Nevertheless, the event can be modeled as an update of the logical copy of memory for that processor.

The third feature, i.e., the memory buffer, represents a significant addition to most previous abstractions, and seems necessary to capture the behavior of many multiprocessor system designs, such as Silicon Graphics and SUN multiprocessors. Again, there may be no physical entity corresponding to this buffer in a real system. The intent of the buffer is to model the scenario where a processor reads the value of its own write before any of the write's sub-operations take effect in memory. This scenario can occur in a cache-coherent multiprocessor if a processor does a write to a location that requires exclusive ownership to be requested and allows a subsequent read (issued by itself) to that location to return the new value while the ownership request is pending.[4] Sindhu et al.'s abstraction [SFC91] models this effect through a conceptual write buffer and allowing the read to return the value of a write before it is retired from this buffer. However, their abstraction is limited in two aspects: they assume the processor stalls on a read operation until a value is returned and they model writes with a single sub-operation. Thus, they do not model out-of-order reads and non-atomic writes.

The abstraction used by Gibbons et al. [GMG91, GM92] to formalize the system requirements for properly-labeled (PL) programs and release consistency meets the criteria we have discussed above; i.e., it models the existence of multiple copies, the non-atomicity of writes, the out-of-order execution of memory operations, and allowing the processor to read its own write before the write is issued to the memory system. (The M_{base} abstraction in [GMG91] is limited because it does not model out-of-order read operations from the same processor; however, the non-blocking M_{base} abstraction in [GM92] removes this restriction.) While their abstraction meets our goals, the methodology used in [GMG91, GM92] to specify system requirements is overconstraining and does not fully expose the range of possible optimizations. We will further discuss their abstraction and framework and contrast it to our approach in Section 4.

The next subsections define the notion of an execution and describe how extra constraints may be placed on the ordering of memory sub-operations in an execution in order to satisfy the semantics of various memory models.

2.2 Definition of an Execution

The abstraction presented in the previous section provides the motivation and intuition for the formalism we present below. Based on this abstraction, the system consists of n processors, $P_1, \ldots, P_n$. We use the same terminology used in the previous section to refer to read and write operations (and sub-operations) to shared-memory. A shared read operation R by P_i is comprised of a single atomic sub-operation, *R(i)*. A shared write operation W by P_i

[4]Such implementations are covered more thoroughly in Section 3.

is comprised of *(n+1)* atomic sub-operations: the initial write sub-operation $W_{init}(i)$ and n sub-operations *W(1)*, ..., *W(n)* (all sub-operations of W access the same location and write the same value).[5]

We define an *execution* of a multiprocessor program as follows. A multiprocessor execution consists of a *set of instruction instances* and a relation called the *program order* on these instruction instances. The program order relation is a partial order on the instruction instances, where the partial order is total for the instruction instances issued by the same processor and is determined by the program text for that processor [SS88]. In addition, the execution consists of a *set of shared-memory sub-operations* that correspond to the instruction instances that access shared-memory. Note that read and write sub-operations to private memory are not included in this set. The notion of an execution is formalized in Definition 1. Below we discuss two conditions that every execution should obey.

The first condition on the execution (Condition 1) is that each processor performs its computation correctly. We refer to this as the uniprocessor correctness condition. Since this work is directed towards specifying multiprocessor behavior, we do not fully formalize a correct uniprocessor, but rely on the intuitive notion. Some of the aspects of correctness we assume involve executing the correct set of instruction instances, returning correct values from private memory, and performing all register operations and local computation correctly. There is a characteristic of the uniprocessor correctness condition as stated in Condition 1 that has a subtle implication on multiprocessor systems. Given an execution E of a correct uniprocessor, if an instruction instance i is in E, then the number of instruction instances in E that are ordered before i by program order should be finite.[6] Effectively, this disallows a processor from issuing instructions that follow (by program order) infinite loops in the program. The exact effect of this requirement on multiprocessor implementations is discussed in Section 3; Appendix D discusses how this requirement may be relaxed to allow more aggressive implementations.

The second condition (Condition 2) is that for every execution, there should exist at least one total order, called the *execution order* (and denoted by $\xrightarrow{xo}$), on all memory sub-operations in the execution that is "consistent" with the values that are returned by the read sub-operations in the execution. The notion of when a total order is "consistent" with the read values is formalized in Condition 2 below. Intuitively, this captures the notion that values returned by read sub-operations should obey the abstraction of the system presented in the previous section.[7] Note that corresponding to each execution, there may be several execution orders that satisfy Condition 2.

Definition 1: Execution
An *execution* of a program consists of the following three components and should obey Conditions 1 and 2 specified below.
(i) A (possibly infinite) set I of instruction instances where each instruction instance is associated with the processor that issued it, registers or memory locations it accessed, and values it read or wrote corresponding to the accessed registers and memory locations.
(ii) A set S of shared-memory read and write sub-operations that contains exactly the following components. For every instruction instance i in I that reads a shared-memory location, the set S contains the memory sub-operation for the reads of i with the same addresses and values as in i. For every instruction instance i in I that writes a shared-memory location, the set S contains the W_{init} sub-operation for the writes of i with the same addresses and values as in i, and possibly other sub-operations of these writes in the memory copies of different processors.
(iii) A partial order, called the program order (denoted $\xrightarrow{po}$), on the instruction instances in I, where the partial order is total for the instruction instances issued by the same processor.

Condition 1: Uniprocessor Correctness Condition
For any processor P_i, the instruction instances of P_i in the set I (including the associated locations and values) belonging to an execution, and its corresponding program order $\xrightarrow{po}$, should be the same as the instruction instances and program order of a conceptual execution of the code of processor P_i on a *correct* uniprocessor, given that the shared-memory reads of P_i (i.e., the R(i)'s) on the uniprocessor are made to return the same values as those specified in the set I.

[5]It is not necessary to explicitly model the W_{init} sub-operation. As will become apparent however, modeling this sub-operation makes it easier to reason about properties such as the value condition (Condition 2).

[6]In general, we assume that for any total or partial order $\xrightarrow{to}$ defined on a (possibly infinite) set K, $\xrightarrow{to}$ orders only a finite number of elements of set K before any element k in the set K, and the remaining (possibly infinite) elements are ordered after k.

[7]To model initial values in the memory copies, we assume there are hypothetical write sub-operations writing the values and for any such write sub-operation W(i) to the same location as a sub-operation X(i) in the execution order, W(i) is before X(i).

Condition 2: Value Condition
Consider the set S of shared-memory read and write sub-operations. If set S is part of an execution, then there must exist a total order called the *execution order* (denoted $\xrightarrow{xo}$) on the memory sub-operations of S such that (a) only a finite number of sub-operations are ordered by $\xrightarrow{xo}$ before any given sub-operation, and (b) any read sub-operation R(i) by P_i returns a value that satisfies the following conditions. If there is a write operation W by P_i to the same location as R(i) such that W_{init}(i) $\xrightarrow{xo}$ R(i) and R(i) $\xrightarrow{xo}$ W(i), then R(i) returns the value of the last such W_{init}(i) in $\xrightarrow{xo}$. Otherwise, R(i) returns the value of W'(i) (from any processor) such that W'(i) is the last write sub-operation to the same location that is ordered before R(i) by $\xrightarrow{xo}$.

Although an execution is defined in terms of a set of instructions and memory sub-operations, we also implicitly associate it with a set of memory operations that comprise the memory sub-operations. We will often refer to this set as memory operations or operations of the execution. We also extend the notion of program order to both sub-operations and operations. Thus, an operation or sub-operation precedes another by program order if their corresponding instructions do so. Further, if an instruction contains multiple memory operations, then we assume that all constituent operations are ordered by $\xrightarrow{po}$ with the exact ordering being defined by the specific instruction. Finally, except in Appendices A and D, the description of system constraints in the paper only concerns the interaction between shared-memory operations. For this reason, we may often ignore the set I of the execution, and use only S and the program order relation on S to describe the execution.

Our definition of execution is influenced by the work by Adve and Hill [AH92] and by Sindhu et al. [SFC91] and combines concepts from both. Adve and Hill define an execution similar to the above except that they do not allow a read, R(i), to return the value of a write whose sub-operation, W(i), occurs after the read in the execution order. On the other hand, Sindhu et al. model write operations as a single sub-operation. However, unlike these two definitions, we chose to include all instructions in the execution as opposed to only shared-memory operations.

From this point on, we assume the above two conditions (1 and 2) are satisfied by all specifications presented in this paper. The next section describes how extra constraints can be placed on an execution to satisfy the semantics of a given memory model.

2.3 Memory Model Specific Requirements

This section describes our methodology for specifying constraints on an execution to reflect memory ordering restrictions that are specific to each memory model. We first discuss general conditions that need to be satisfied by an execution and an implementation in order to maintain the semantics of a memory model. We then present specific system constraints for satisfying sequential consistency and other more relaxed memory models.

We use the following terminology below. We denote program order by $\xrightarrow{po}$. Two operations (or sub-operations) conflict if both are to the same location and at least one is a write [SS88]. We use R (or Ri, Rj, Rk, etc.) and W (or Wi, Wj, Wk, etc.) to denote any read and write operations respectively. RW denotes either a read or a write. For RCpc, we use R_acq and W_rel to denote acquire and release operations, respectively.

The conditions for each model consist of three parts: the initiation condition (for memory sub-operations), the $\xrightarrow{sxo}$ condition, and the termination condition (for memory sub-operations). Each of these parts specifies a restriction on the execution order relation. For a system to obey a memory model, any execution allowed by the system must have an execution order that satisfies each of the above three parts as specified for the corresponding model. We discuss each of these parts below.

The initiation condition for memory sub-operations is a restriction on the order of the initial write sub-operations (i.e., W_{init}(i)) with respect to other sub-operations. Its main purpose is to capture the program order among read and write sub-operations to the same location.

Condition 3: Initiation Condition (for memory sub-operations)
Given two operations by P_i to the same location, the following must be true. If R $\xrightarrow{po}$ W, then R(i) $\xrightarrow{xo}$ W_{init}(i). If W $\xrightarrow{po}$ R, then W_{init}(i) $\xrightarrow{xo}$ R(i). If W1 $\xrightarrow{po}$ W2, then $W1_{init}$(i) $\xrightarrow{xo}$ $W2_{init}$(i).

The $\xrightarrow{sxo}$ condition imposed on the execution order is based on the $\xrightarrow{sxo}$ relation ($\xrightarrow{sxo}$ is a *subset* of $\xrightarrow{xo}$, thus the name) among memory operations that is defined for each model (examples of this follow in the next section). This condition requires certain sub-operations to execute in the same order as that specified by $\xrightarrow{sxo}$ on their corresponding operations. The formal description follows.

Condition 4: $\xrightarrow{sxo}$ Condition
Let X and Y be memory operations. If X $\xrightarrow{sxo}$ Y, then X(i) must appear before Y(j) in the execution order for some i,j pairs (the i,j pairs used are specific to each model). The $\xrightarrow{sxo}$ relation is individually specified for each memory model.

The termination condition for memory sub-operations requires that certain write sub-operations take effect in finite "time". The subset of write sub-operations that obey this condition is specified by each memory model. The condition is as follows.

Condition 5: Termination Condition (for memory sub-operations)
Suppose a write sub-operation W_{init}(i) (belonging to operation W) by P_i appears in the execution. The termination condition requires that the other *n* corresponding sub-operations, *W(1) ... W(n)*, appear in the execution as well. A memory model may restrict this condition to a subset of all write sub-operations.

We define a *valid execution order* as one that satisfies the constraints specified by a memory model. A *valid execution* is then an execution that has a corresponding valid execution order.[8] A correct implementation for a model can then be defined as one that allows valid executions only. The definitions for these terms are presented below.

Definition 2: Valid Execution Order
An *execution order* is *valid* for a memory model iff it satisfies the conditions (Conditions 3, 4, and 5) specified for that model.

Definition 3: Valid Execution
An execution is a *valid execution* under a memory model iff *at least one* valid execution order can be constructed for the execution.

Definition 4: Correct Implementation of a Memory Model
A system correctly implements a memory model iff *every* execution allowed by the system is a *valid* execution for that model.

Typical implementations obey the constraints on execution order directly (and usually conservatively); i.e., constraints on the execution order are enforced among sub-operations in real time. However, a system designer is free to violate the constraints imposed by a specific memory model as long as the resulting executions *appear* to obey these constraints.[9] While this provides room for flexibility, in practice, we believe it is difficult for a designer to come up with less restrictive constraints and guarantee that the resulting executions will be the same as with the original constraints. Therefore, it is important to initially provide the designer with as minimal a set of constraints as possible.

To avoid execution order constraints that are unnecessary for maintaining the semantics of a model, we exploit the following observation in our framework. Given a valid execution order $\xrightarrow{xo}$, any execution order $\xrightarrow{xo'}$ that maintains the same order among conflicting sub-operations as $\xrightarrow{xo}$ is also a valid execution order. The reason is that the order among conflicting sub-operations completely determines the values read and written by the sub-operations. Therefore, to specify valid execution orders, it is sufficient to constrain the execution order among conflicting sub-operations only. Specifically, the use of this observation in the $\xrightarrow{sxo}$ condition is the primary distinguishing feature of our methodology. (It is possible that two execution orders may impose different orders among conflicting sub-operations and yet result in the same values being read and written by the sub-operations; however, exploiting this observation to provide the minimal possible constraints on the execution orders seems difficult.)

[8]Note that our framework covers infinite executions since an execution is a set which can be infinite, and the conditions for different models are specified only as restrictions on this set. In practice, on a real system, one may want to test whether a partial execution (potentially leading to an infinite execution) generates a valid outcome. For this, we can define a partial execution as a set I' of instructions, a set S' of sub-operations, and a program order relation that represent the instructions and sub-operations that have been generated so far in the system and the corresponding program order. Then the partial execution generates a valid outcome or is valid if it could possibly lead to a valid complete execution; i.e., a partial execution is valid if its sets I' and S' are respectively subset of sets I and S of the instructions and sub-operations of some valid execution. Note here that in forming I' and S', we should not include speculative instructions or sub-operations that may have been generated but are not yet committed in the system.

[9]This would not be possible for models such as linearizability [HW90] where correctness depends on the real time occurrence of events. However, for the models we have been discussing, the real time occurrence of events is assumed to be unobservable.

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if one of

1. X $\xrightarrow{po}$ Y
2. X $\xrightarrow{co}$ Y
3. atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$**:**

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(j) for all i,j.
Termination condition holds for all sub-operations.

Figure 2: Conservative conditions for SC.

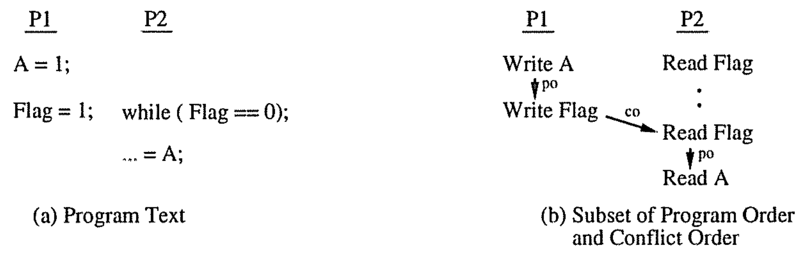
The above observation has been previously made by others as well. For example, Shasha and Snir [SS88] exploit a similar observation in identifying a minimal set of orders (derived from the program order) that are sufficient for achieving sequential consistency for a given program. Collier [Col92] also uses this observation for proving equivalences between different sets of ordering constraints. However, previous specifications of memory models have not exploited this observation to its full potential. Specifically, many of the specifications impose unnecessary ordering constraints on non-conflicting pairs of memory operations; even Shasha and Snir's implementation involves imposing delays among non-conflicting memory operations that occur in program order. In contrast, our framework presents a unified methodology for specifying ordering constraints that apply to pairs of conflicting memory operations only. We next present the constraints for sequential consistency in our framework and contrast the two approaches of specifying the execution order constraints among both conflicting and non-conflicting sub-operations versus among conflicting sub-operations only. We then proceed with specifications for other more relaxed memory models.

2.3.1 Requirements for Sequential Consistency

This section develops the system requirements for maintaining SC. We first develop a set of sufficient constraints on the execution order and then show how our framework allows us to express a substantially weakened set of constraints that still satisfy SC.

First, we define some terminology below. As before, we denote the program order by $\xrightarrow{po}$ and the execution order by $\xrightarrow{xo}$. A condition like "X(i) $\xrightarrow{xo}$ Y(j) for all i,j" implicitly refers to pairs of values for i and j for which both X(i) and Y(j) are defined [AH92]. We also implicitly assume that X(i) and Y(j) appear in the execution for all such pairs. For example, given R is issued by P_k, "W(i) $\xrightarrow{xo}$ R(j) for all i,j" reduces to "W(i) $\xrightarrow{xo}$ R(k) for all i" because R is not defined for any value other than k. As mentioned before, two operations *conflict* if both are to the same location and at least one is a write [SS88]. Two sub-operations conflict if their corresponding operations conflict with one another. The *conflict order* relation (denoted $\xrightarrow{co}$) is defined as follows. For an execution order $\xrightarrow{xo}$ and two conflicting operations X and Y, X $\xrightarrow{co}$ Y iff X(i) $\xrightarrow{xo}$ Y(i) holds for some i. If X and Y are on different processors and X $\xrightarrow{co}$ Y, then X and Y are also ordered by the interprocessor conflict order X $\xrightarrow{co'}$ Y. Since atomic read-modify-write operations (e.g., test-and-set and fetch-and-increment) are common in shared-memory machines, we include them in our framework as well. Given an atomic read-modify-write operation, RMW denotes the operation itself and AR and AW denote the individual read and write operations, respectively. We assume AR $\xrightarrow{po}$ AW.

Lamport [Lam79] defines *sequential consistency* as follows: 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". While this definition specifies the possible outcomes for a program, it does not directly translate into an implementation of SC. Therefore, to aid designers, we need to specify a set of conditions that more closely relate to an implementation. For example, Dubois et al. [SD87] have shown that to satisfy SC, it is sufficient to maintain program order and make writes appear atomic in an implementation. Figure 2 shows these conditions expressed in our framework.



```
P1            P2

A = 1;

Flag = 1;     while ( Flag == 0);

              ... = A;
```

Figure 3: Example code segment.

Conservative conditions for SC in our framework

Figure 2 gives conservative conditions for SC in our framework. The $\xrightarrow{sxo}$ relation ($\xrightarrow{sxo}$ is a subset of $\xrightarrow{xo}$) captures some of the conditions used to constrain the execution order $\xrightarrow{xo}$. The figure shows three conditions under $\xrightarrow{sxo}$. The first condition under $\xrightarrow{sxo}$ captures the program order between operations from the same processor. The second condition captures the conflict order between conflicting operations (related to making writes appear atomic). And finally, the third condition captures the atomicity of read-modify-write operations with respect to other writes to the same location. (Read-modify-write operations are usually not covered in other frameworks in the literature.) Given the definition of $\xrightarrow{sxo}$, the constraints on the execution order $\xrightarrow{xo}$ are as follows: (i) the initiation condition holds, (ii) if two operations are ordered by $\xrightarrow{sxo}$, then all sub-operations of the first operation are ordered by $\xrightarrow{xo}$ before any sub-operation of the second operation, and (iii) the termination condition holds among all sub-operations.

To better understand the conservative conditions, consider the example code segment in Figure 3. The left side of the figure shows a program where one processor writes a location and sets a flag, while the other processor waits for the flag to be set and then reads the location. Assume the values of A and Flag are initially zero. The right side of the figure shows the program order arcs ($\xrightarrow{po}$) and the conflict order arcs ($\xrightarrow{co}$) between the write and the successful read of the flag. With sequential consistency, the read of A should return the value 1 in every execution. Let us consider how the conditions on the execution order in Figure 2 ensure this. The program order condition constrains $\xrightarrow{xo}$ as follows: $W_A(i) \xrightarrow{xo} W_{Flag}(j)$ and $R_{Flag}(i) \xrightarrow{xo} R_A(j)$ for all i,j. The conflict order condition constrains $\xrightarrow{xo}$ such that $W_{Flag}(i) \xrightarrow{xo} R_{Flag}(j)$ for all i,j. By transitivity, we have $W_A(i) \xrightarrow{xo} R_A(j)$ for all i,j, which denotes that all sub-operations of the write to A happen before the read A sub-operation, ensuring that the read returns the value 1 in all executions. It is possible to weaken the conditions in Figure 2 while maintaining the semantics of SC; we explore this more aggressive set of conditions next.

Aggressive conditions for SC in our framework

The conditions in Figure 2 impose constraints on the execution order between both conflicting and non-conflicting sub-operations. The key insight behind the more aggressive conditions for SC is to impose constraints on the execution order *among conflicting sub-operations only.* Figure 4 shows these aggressive conditions for SC. As before, the $\xrightarrow{sxo}$ relation captures the conditions that constrain the execution order. We have added two other relations, $\xrightarrow{spo}$ and $\xrightarrow{sco}$, for notational convenience ($\xrightarrow{spo}$ is a subset of $\xrightarrow{po}$ and $\xrightarrow{sco}$ is a subset of the transitive closure of $\xrightarrow{co}$). These relations capture certain $\xrightarrow{po}$ and $\xrightarrow{co}$ orders that are used in the $\xrightarrow{sxo}$ relation. Note that $\xrightarrow{sxo}$ is only defined among conflicting operations. Given $\xrightarrow{sxo}$, the constraints on $\xrightarrow{xo}$ are as follows: if X $\xrightarrow{sxo}$ Y (X and Y conflicting), then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Let us now consider the different conditions for the $\xrightarrow{sxo}$ relation. The first condition under $\xrightarrow{sxo}$ is uniprocessor dependence and captures the order of operations from the same processor and to the same location. Together with the initiation condition (Condition 3) and the value condition (Condition 2), this condition ensures that a read returns either the value of the last write to the same location that is before it in program order or the value of a subsequent write to that location. The second condition is coherence which ensures that write sub-operations to the same location take effect in the same order in every memory copy. The third set of conditions, denoted as the multiprocessor dependence chain, captures the relation among conflicting operations that are ordered by $\xrightarrow{po} \cup \xrightarrow{co}$. As we will see shortly, this set of conditions is the key component of our aggressive specification that differentiates it from the conservative conditions (as in Figure 2). The notation "$\{A \xrightarrow{sco} B \xrightarrow{spo}\}+$" denotes

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X and Y are to *different* locations and X $\xrightarrow{po}$ Y

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co'}$ Y

R $\xrightarrow{co'}$ W $\xrightarrow{co'}$ R

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co'}$ R $\xrightarrow{po}$ R

RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$**:**

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all sub-operations.

Figure 4: Aggressive conditions for SC.

one or more occurrences of this pattern within the chain.[10] Finally, the condition on atomic read-modify-writes is the same as in the conservative conditions. The formal proof of equivalence between these aggressive conditions and Lamport's definition of SC appears in Appendix C.

Consider the program in Figure 3 again, this time with the aggressive conditions. The relevant condition in Figure 4 that ensures the correct behavior for this program is the second condition under the multiprocessor dependence chain category. This condition applies as follows: $W_A \xrightarrow{sxo} R_A$ because $W_A \xrightarrow{spo} W_{Flag} \xrightarrow{sco} R_{Flag} \xrightarrow{spo} R_A$. In turn, $W_A \xrightarrow{sxo} R_A$ constrains the execution order among the sub-operations such that W_A(i) $\xrightarrow{xo}$ R_A(i) for all i (this reduces to $W_A(2) \xrightarrow{xo} R_A(2)$ since R_A(i) is only defined for i=2). Therefore, every execution is required to return the value of 1 for the read of A. Note that there are many more valid execution orders possible under the aggressive conditions as compared to the conservative conditions. For example, the following execution order, $W_{Flag}(2) \xrightarrow{xo} R_{Flag}(2) \xrightarrow{xo} W_A(2) \xrightarrow{xo} R_A(2)$, is allowed under the aggressive but not under the conservative conditions (because program order is violated on the first processor).

Even though the aggressive constraints place much fewer constraints on the execution order, the resulting executions *"appear the same as if"* the strict conditions were maintained. While the difference between the two sets of conditions is not discernable by the programmer, from a system design perspective, the aggressive conditions allow more flexibility with respect to optimizations that are possible. For example, more aggressive implementations of SC such as the lazy caching technique [ABM89] do not directly satisfy conditions in Figure 2, but can be shown to directly satisfy the aggressive conditions in Figure 4. We further explore such implications on system implementation in Section 3.

2.3.2 Requirements for Other Models

The previous section presented system requirements specific to sequential consistency. However, the framework we described is general and can be used to express the system requirements for other memory models as well. While the conditions for various models differ in the specific execution orders they allow, the basic categories of conditions captured by the $\xrightarrow{sxo}$ relation and the way $\xrightarrow{sxo}$ is used to constrain the execution order among conflicting sub-operations is uniform across many of the specifications. We present the conditions for one of the hardware-centric models, RCpc, in this section. The conditions for several other relaxed models in addition to proofs of their correctness are provided in Appendices B and C. We have also used this framework for specifying

[10]This is similar to regular expression notation. We also use "*" to denote zero or more occurrences of a pattern.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$:

X $\xrightarrow{spo'}$ Y if X,Y are the first and last operations in one of

RW $\xrightarrow{po}$ W_rel

R_acq $\xrightarrow{po}$ RW

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo'}\}+$ Y

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for W_rel sub-operations.

Figure 5: Aggressive conditions for RCpc.

sufficient system requirements for programmer-centric models such as PLpc [AGG+93].

Figure 5 presents the conditions for RCpc [GLL+90, GGH93] in their aggressive form where the constraints on the execution order are present among conflicting sub-operations only.[11] We use R_acq and W_rel to denote a read acquire and a write release, respectively, and R and W still denote any read or write, including acquires and releases. Note that even though we are representing a different model than SC, the way the conditions are presented is quite similar to the conditions for SC in Figure 4. Of course, at the detailed level, the exact orders that are maintained do differ. As before, we use the $\xrightarrow{spo}$ and $\xrightarrow{sco}$ relations for notational convenience in order to capture the relevant $\xrightarrow{po}$ and $\xrightarrow{co}$ orders that are used in the $\xrightarrow{sxo}$ conditions. Consider the $\xrightarrow{spo}$ relation first. While for SC, $\xrightarrow{spo}$ holds between any two operations ordered by $\xrightarrow{po}$, in RCpc, $\xrightarrow{spo}$ holds for only a subset of the operations ordered by $\xrightarrow{po}$: (i) if the first operation is an acquire, or (ii) if the second operation is a release, or (iii) if the two operations are related by the reach relation $\xrightarrow{rch}$ (will be discussed shortly), or (iv) the transitive closure of the above. Similarly, R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 does not constitute an $\xrightarrow{sco}$ for RCpc unless R1 and R2 are on the same processor. The conditions under $\xrightarrow{sxo}$ are also similar in form to those of SC except the third chain under multiprocessor dependence is not present in the conditions for RCpc. The manner in which the $\xrightarrow{sxo}$ relation is used to constrain $\xrightarrow{xo}$ is also identical to SC. However, the constraints imposed by the termination condition apply only to release sub-operations under RCpc.[12]

Consider our example code segment in Figure 3 with the conditions for RCpc. Assume the write to Flag is identified as a release and the read of Flag is identified as an acquire. Therefore, we have $W_A \xrightarrow{spo} W_{Flag}$ and $R_{Flag} \xrightarrow{spo} R_A$. In addition, $W_{Flag} \xrightarrow{co} R_{Flag}$ translates to $W_{Flag} \xrightarrow{sco} R_{Flag}$. By the second condition under multiprocessor dependence chain, we have $W_A \xrightarrow{sxo} R_A$ since $W_A \xrightarrow{spo} W_{Flag} \xrightarrow{sco} R_{Flag} \xrightarrow{spo} R_A$, which in turn imposes the constraint $W_A(i) \xrightarrow{xo} R_A(i)$ for all i. Therefore, given the read and write operations to Flag are labeled as release and acquire, respectively, the read of A will always return the value 1. However, consider the

[11]For simplicity, we treat all read and write nsync operations [GLL+90] as acquires and releases, respectively. Appendix B shows the conditions without this simplification.

[12]This termination condition is sufficient for properly-labeled programs (PL [GLL+90] and PLpc [GAG+92]) where all races are identified. For programs that do not identify all races (e.g., chaotic relaxation), it may be desirable to prescribe time intervals at which termination is upheld for all writes (e.g., writes may be forced to take effect in all memory copies every so many cycles).

case, for example, where the write to Flag is not labeled as a release. In this case, $W_A \xrightarrow{spo} W_{Flag}$ no longer holds, and none of the conditions in RCpc constrain the execution order between operations to location A any more. Therefore, $R_A(2) \xrightarrow{xo} W_A(2)$ is an allowed execution order and the read of A may return the value of 0 in some executions (an analogous situation occurs if the read of Flag is not labeled as an acquire).

We now describe the reach relation ($\xrightarrow{rch}$) that is used as part of the $\xrightarrow{spo}$ definition in Figure 5. The reach relation captures the interaction between memory operations that arise due to uniprocessor control and data dependences in each process' program and extends these dependences to include multiprocessor specific interactions as well (e.g., "dependence" implied by the $\xrightarrow{spo}$ relation). To motivate this, consider the code example in Figure 6. Here, we show P1 reading location A, testing its value, and conditionally writing location B, while P2 does the symmetric computation with accesses to A and B interchanged. Assume all memory locations are initialized to zero. In each processor, whether the write occurs depends on the value returned by the read. Assume that each processor waits for its read to take effect before allowing its write to take effect in the other processor's memory copy. In such an implementation, the writes do not appear in the execution of either processor since both conditionals fail. Now consider an implementation that speculatively allows writes to occur before the result of the conditional is resolved (e.g., possible in systems using branch prediction with speculative execution past branches). In such an implementation, it is possible to get an execution in which both conditionals succeed and both writes occur in the execution. However, the above execution is somewhat anomalous since writes in the future affect the value of previous reads that in turn determine whether the writes should have been executed at all. This type of anomalous behavior is automatically eliminated in models such as SC, PC, TSO, and PSO that maintain the $\xrightarrow{spo}$ relation from a read to writes that follow it in program order. On the other hand, such anomalous executions could occur with models such as RCsc, RCpc, and WO if the $\xrightarrow{rch}$ relation is not included in $\xrightarrow{spo}$. However, these models are intended to appear SC to a class of well-behaved programs [AH90b, GLL+90, AH93, GAG+92], and executions such as the one described above violate this intent. Therefore, it is necessary to give precise conditions that would prohibit the above type of anomalous executions.

The main purpose for the reach condition is to disallow the types of anomalous executions that arise if we allow "speculative" write sub-operations to take effect in other processors' memory copies. In most previous work, such conditions were either implicitly assumed or assumed to be imposed by informal descriptions such as "intra-processor dependencies are preserved" [AH90b] or "uniprocessor control and data dependences are respected" [GLL+90]. Some proofs of correctness (e.g., proof of correctness for PL programs executing on the RCsc model [GLL+90]) formalized certain aspects of this condition, but the full condition was never presented in precise terms. Recent work by Adve and Hill specifies this condition more explicitly in the context of the DRF1 model [AH92] and proves that it is sufficient for ensuring sequentially consistent results to data-race-free programs on models such as WO and RCsc. More recently, we have jointly formalized an aggressive form of this condition as part of specifying sufficient conditions for PLpc [AGG+93] and have proven that PLpc programs provide sequentially consistent results on models such as RCpc. Our use of the $\xrightarrow{rch}$ relation for WO, RCsc, and RCpc is a straightforward adaptation of the condition developed for PLpc. The formal proof that this type of a condition is sufficient for providing sequentially consistent results for a general class of programmer-centric models will appear in [Adv93].

Intuitively, a read operation reaches a write operation that follows it in program order ($R \xrightarrow{rch} W$) if the read determines whether the write will execute, the address accessed by the write, or the value written by it.[13] In addition, $R \xrightarrow{rch} W$ if the read controls the execution or address of another memory operation that is before W in program order and is related to W by certain program order ($\xrightarrow{po}$) arcs (e.g., $\xrightarrow{spo}$) that are used in the $\xrightarrow{sxo}$ relation.[14] The formal definition of the $\xrightarrow{rch}$ relation is provided in Appendix A and is a straightforward adaptation of the conditions developed for PLpc [AGG+93]. This formulation represents the most aggressive condition developed so far and allows for several optimizations while disallowing the anomalous executions (e.g., speculative reads are allowed; writes that do not depend on the outcome of a branch are allowed to execute before the branch is resolved).

Compared to the original conditions for RCpc [GLL+90, GGH93], the conditions presented here are more complete (with respect to precisely specifying data and control dependences) and expose more aggressive opti-

[13]Similar to the uniprocessor correctness condition, the reach condition requires considering all instructions (not just memory instructions) to determine whether the relation holds between a pair of memory operations.

[14]This aspect of the reach relation makes it memory model dependent since different models have different $\xrightarrow{spo}$ relations.

```
P1                    P2

if (A == 1) {         if (B == 1) {
   B = 1;                A = 1;
}                     }
```

Figure 6: Example to illustrate the reach condition.

mizations while maintaining the same semantics. We have provided the specification for several other models in Appendix B and in each case, the new specification exposes more aggressive optimizations than the original one. Similar to RCpc, the new specifications for WO and RCsc are also more complete. The next section will further discuss how exposing more optimizations expands the range of efficient implementations for each model.

3 Implementation Techniques

This section describes how to implement a memory model that is specified within our framework. The first part describes issues related to the architecture (i.e., hardware and any runtime software) while the second part describes compiler-related issues.

3.1 Architectural Issues

This section discusses implementing the system requirements specified by our methodology at the architectural level. We assume the uniprocessor correctness condition (Condition 1) is trivially satisfied.[15] The remaining requirements consist of constraints on the execution order due to the the value, initiation, $\xrightarrow{sxo}$, and termination conditions (Conditions 2-5). We use the following terminology below. A write operation W by P_i whose W$_{init}$(i) sub-operation appears in the execution but whose W(i) sub-operation is yet to appear is referred to as a *pending* write. We will use the term *uniprocessor dependence condition* to imply the constraint imposed on the execution order due to the uniprocessor dependence component of the $\xrightarrow{sxo}$ condition. We will use the terms *coherence condition*, *multiprocessor dependence chain condition*, and *read-modify-write condition* similarly, and will discuss each of these conditions separately. Of all the conditions discussed below, the multiprocessor dependence chain condition is the most important contribution of this work since it allows a wider range of optimizations and implementations than previous specifications by imposing constraints among conflicting operations only.

3.1.1 Value, Initiation, Uniprocessor Dependence, and Coherence Conditions

The value, initiation, uniprocessor dependence, and coherence conditions interact in a close way and so we discuss them together. The intent of the value, initiation, and uniprocessor dependence conditions is to maintain the intuitive notion of uniprocessor data dependences; i.e., a processor should "see the effect of" its own conflicting operations in program order. An intuitive way of ensuring uniprocessor data dependence would be to require that the sub-operations of two conflicting operations of a single processor should execute in program order (e.g., if R $\xrightarrow{po}$ W on P_i, then R(i) $\xrightarrow{xo}$ W$_{init}$(i) and R(i) $\xrightarrow{xo}$ W(i)). With our methodology, for SC and RCpc, the initiation and uniprocessor dependence conditions (Figures 4 and 5) ensure the above for two conflicting writes in program order (i.e., W1 $\xrightarrow{po}$ W2 on P_i implies W1$_{init}$(i) $\xrightarrow{xo}$ W2$_{init}$(i) and W1(i) $\xrightarrow{xo}$ W2(i)) and for a write following a conflicting read in program order. However, if a read R follows a write W in program order on P_i, the uniprocessor condition does not require W(i) $\xrightarrow{xo}$ R(i). While this may imply a possible violation of uniprocessor data dependence, the initiation condition requires W$_{init}$(i) $\xrightarrow{xo}$ R(i) and even if R(i) $\xrightarrow{xo}$ W(i) (i.e., write is pending), the value condition still ensures that R will return the value of W (or a later conflicting write that is before R by program order). This ensures that the essential notion of uniprocessor data dependence is maintained; i.e., a read sees the effect of the conflicting writes that are before it in program order. Below, we first describe how the value, initiation, uniprocessor dependence, and coherence conditions can be implemented, and

[15]To satisfy Condition 1, one of the requirements is that operations following an unbounded loop (by program order) should not be executed until it is known that the loop will terminate. Appendix D discusses how this restriction may be relaxed in certain cases.

```
         P1          P2
    (a)  A = 1       A = 2   (b)
                     x = A   (c)
```

Figure 7: Example code segment showing interaction between the various conditions.

then discuss their interaction which requires our less intuitive form for the value and uniprocessor dependence conditions.

Implementation of value, initiation, uniprocessor dependence, and coherence conditions

A typical way in which the value, initiation, and uniprocessor dependence conditions are maintained is as follows (all sub-operations below are from the same processor P_i). A write sub-operation is delayed until previous read and write sub-operations to the same location have executed in the memory copy of the issuing processor (i.e., R $\xrightarrow{po}$ W implies R(i) $\xrightarrow{xo}$ W_{init}(i) and R(i) $\xrightarrow{xo}$ W(i); W1 $\xrightarrow{po}$ W2 implies $W1_{init}$(i) $\xrightarrow{xo}$ $W2_{init}$(i) and W1(i) $\xrightarrow{xo}$ W2(i)).[16] Read sub-operations do not need to be delayed for previous (in program order) read and write sub-operations (except W $\xrightarrow{po}$ R implies W_{init}(i) $\xrightarrow{xo}$ R(i)), but if there are pending memory write sub-operations to the same location, the read has to return the value of the pending write to the same location that is the last one before it in program order. For example, in an implementation that uses write buffers, the appropriate value is forwarded to the read if there is a conflicting write in the buffer. The above techniques are well understood since similar conditions need to be maintained in uniprocessors.

The coherence condition constrains write sub-operations to the same location to occur in the same order with respect to every memory copy (i.e., W1(i) $\xrightarrow{xo}$ W2(i) for all i). This is typically achieved through a logical serialization point per location where writes to that location are serialized (e.g., at the home directory in a directory-based coherence protocol). Techniques for maintaining coherence are also well understood and have been covered in the literature (e.g., see [GLL+90, LLG+90] for a description of the Dash coherence protocol).

Interaction between value, initiation, uniprocessor dependence, and coherence conditions

There is a subtle interaction between the value, initiation, uniprocessor dependence, and coherence conditions when all of them are to be satisfied by operations to a location. Consider the code segment in Figure 7. Assume all values are initially 0. Both P1 and P2 write to location A and P2 also reads this location. Let W1 denote the write on P1 and W2 the write on P2. Coherence requires all processors to observe the two writes of A in the same order. As discussed above, this is usually achieved by sending the two writes to a serialization point from where their sub-operations are sent to different memory copies in the same order. Thus, the write sub-operation of P2 (i.e., W2(2)) cannot execute in P2's memory copy as soon as it is issued. Instead, it has to be sent to a serialization point and is executed in P2's memory copy only after all other write sub-operations to A that are serialized before this write have executed in P2's memory copy. For our case, assume that P1's write is serialized before P2's write. Now consider the state when P2's write is waiting to be serialized and P2 issues its read of A. For the traditional notion of data dependences to be satisfied, this read should return the value of P2's write on A or a write that is serialized after P2's write. Thus, to satisfy data dependences, it is always safe for P2's read to return the value of P2's write. The key question that determines the form of our conditions is: *can this read return the value of P2's write before the write is serialized, and hence possibly before P1's and P2's write sub-operations (W1(2) and W2(2)) are executed in P2's memory copy?* The answer to this question, and the form of the uniprocessor, initiation, and value conditions, depends on the memory model.

For models which do not allow the read to return the value of P2's write before it is serialized, we do not need to model the W_{init}(i) sub-operation and we do not need an explicit value condition (i.e., the value condition would simply require a read to return the value of the last write to the same location that is before it in the execution order). Instead, we can make the uniprocessor dependence condition require that for a read following a write in program order, the read sub-operation should appear after the write sub-operation in the execution order. However, such models impose a delay on the read since it has to wait for a previous write to be serialized.

[16]The coherence condition discussed below ensures that the sub-operations of two conflicting writes of a processor execute in the correct relative order in memory copies of other processors.

Other models (including all models discussed in this paper) can eliminate the above delay by allowing P2's read to return the value of P2's write before the write is serialized. However, with this optimization, the write sub-operation in P2's memory copy appears after the read in the execution order, and thus we cannot require the uniprocessor dependence condition to include a constraint on the order of execution of a read following a write. Instead, the traditional notion of data dependence is maintained by using the initiation and value conditions. These two conditions ensure that even if a write that is before a read by program order is executed after the read, the read will still see the effect of the write. In particular, if a write that is before a read by program order appears after the read in the execution order (i.e., R(i) $\xrightarrow{xo}$ W(i) even though W_{init}(i) $\xrightarrow{xo}$ R(i)), then the read will return the value of the last conflicting write ordered before it by program order. For example, referring back to Figure 7, let W1(2) and W2(2) denote the sub-operations for the two writes (by P1 and P2) destined for P2's memory copy and let R(2) denote the sub-operation for P2's read. Assume W1(2) $\xrightarrow{xo}$ W2(2) by coherence, as we did above. The uniprocessor condition for the models that allow the read to occur while the write is pending corresponds to allowing R(2) $\xrightarrow{xo}$ W2(2). Then one possible execution order is R(2) $\xrightarrow{xo}$ W1(2) $\xrightarrow{xo}$ W2(2). The value and initiation conditions for such models ensure that the value returned by R(2) is that of W2(2), and not of any write that is before W2(2), thereby maintaining data dependence.

Thus, our notions of uniprocessor dependence, initiation, and value conditions precisely and directly capture the optimization of allowing a read to return the value of its own processor's write before it is serialized. This optimization is important since it is used by commercial multiprocessors such as those by Silicon Graphics and SUN, and many of the memory models considered in this paper allow it (for SC, this had not been explicitly stated in previous work).

Often, the above optimization of allowing a read to return the value of a write that is not yet serialized is implemented in the form described by our abstraction; i.e., through early reads from the write buffer. An alternate form of implementing this optimization is as follows. A write is allowed to update the logical memory copy of the issuing processor before it is serialized. A read in this case simply returns the value in the processor's memory copy. However, care must be taken in handling the conflicting write sub-operations that arrive from other processors while the serialization of the current write is pending.[17] This typically involves ignoring any write sub-operations (e.g., invalidation or update messages) that arrive for this location while this processor's write is still awaiting its response from the serialization point.[18] Therefore, it effectively appears as if writes that were serialized before this processor's write took effect before it in this processor's memory copy. Some systems like DASH [LLG+90] use a hybrid scheme that is a merge of the two techniques discussed above. In DASH, a write updates the write-through first level cache immediately (second technique), but the update to the second level cache is delayed by the write-buffer until ownership is granted (first technique).

3.1.2 Multiprocessor Dependence Chain Condition

The multiprocessor dependence chain condition captures the relation among conflicting operations that are ordered by certain program order and conflict order arcs. This condition is a key contribution of this work since it allows a much wider range of optimizations and implementations than most previous specifications. The strength of our framework is that it makes this wide range of implementations relatively obvious, and consequently, simplifies the task of verifying if an aggressive implementation obeys a memory model.

Below, we first discuss the more conservative (and consequently simpler) implementations for satisfying the multiprocessor dependence chain condition. Then we follow with more aggressive (and consequently more difficult) implementations that exploit the extra flexibility offered by our specification approach. The discussion below focuses on implementing the multiprocessor dependence conditions for SC (Figure 4). However, the implementation techniques apply to specifications for the other models as well.

Conservative implementations of the multiprocessor dependence chain condition

The first multiprocessor dependence chain for SC is of the form W $\xrightarrow{co}$ R1 $\xrightarrow{po}$ R2, where all three operations are to the same location. Assume R1 and R2 are issued by P_i. Given such a chain, the condition on the execution

[17] We assume sub-operations to the same location arrive in order from the coherence serialization point.

[18] Invalidations may apply to more than one location since they are typically at a cache line granularity. In this case, the specific locations that are pending should not be invalidated while other locations in the cache line need to be invalidated. This can be achieved by keeping valid bits per word, for example.

order is W(i) $\xrightarrow{xo}$ R2(i). This condition can be trivially satisfied by maintaining program order among reads to the same location. In other words, the implementation ensures that if R1 $\xrightarrow{po}$ R2 on P_i and both are to the same location, then R1(i) $\xrightarrow{xo}$ R2(i).

The second and third conditions under multiprocessor dependence comprise of chains of $\xrightarrow{spo}$ and $\xrightarrow{sco}$ arcs which represent a subset of the program order ($\xrightarrow{po}$) and conflict order ($\xrightarrow{co}$), respectively. Given two conflicting operations X and Y at the beginning and end of such a chain, the execution order is required to obey X(i) $\xrightarrow{xo}$ Y(i) for all i. The simple way to maintain this order is to enforce an order between the consecutive operations in the chain. This is similar to how the conservative conditions for SC (shown in Figure 2) implicitly satisfy this condition. For consecutive operations related by $\xrightarrow{spo}$, we can maintain the program order such that A $\xrightarrow{spo}$ B ensures A(i) $\xrightarrow{xo}$ B(j) for all i,j. This can be achieved by delaying sub-operations of B until all sub-operations of A are completed. ([GLL+90] gives a set of fence operations that provide one general way of achieving this delay; Appendix A further discusses the particular case when the delay is due to the $\xrightarrow{rch}$ component of $\xrightarrow{spo}$.) The above is sufficient to maintain the required order for chains that do not begin with W $\xrightarrow{sco}$ R (e.g., the third multiprocessor dependence chain for SC) and do not contain an $\xrightarrow{sco}$ of the form R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2. For chains that begin with W $\xrightarrow{sco}$ R or comprise of $\xrightarrow{sco}$ of the form R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 (where R1 and R2 are on different processors), the additional sufficient requirement is to make the writes that occur in such $\xrightarrow{sco}$'s appear atomic. One way of making a write appear atomic is to ensure that no read is allowed to return the value of the write until all old copies of the line being written are either invalidated or updated with the new value [SD87, GLL +90].

Aggressive implementations of multiprocessor dependence chain condition

The primary reason our aggressive conditions lead to more flexible implementations is because we constrain the execution order only among conflicting accesses. This is most important for the specification of the orders implied by the multiprocessor dependence chain. Conservative conditions (e.g., Figure 2) typically specify the overall order by requiring order between all intermediate operations in the chain; the simple implementation of the multiprocessor dependence condition that we described above maintains the order in a similar way. However, maintaining the order between intermediate operations is not necessary for correctness. Below, we discuss implementations proposed in the literature that maintain the multiprocessor dependence condition more aggressively. We discuss the first implementation in some detail, and only give the key ideas for the others.

The first implementation is the lazy caching technique proposed by Afek et al. [ABM89]. This implementation uses a bus-based update protocol to keep the caches coherent. Sequential consistency is maintained in an aggressive way as explained below. When a processor issues a write, it only waits for the updates of its write to be queued into the FIFO input queues of the other processors and can continue with its next operation before the updates actually take effect in the caches. Therefore, the implementation does not directly satisfy the conservative conditions for SC specified in Figure 2 or the conditions specified by Dubois et al. [SD87], since X $\xrightarrow{po}$ Y does not necessarily imply X(i) $\xrightarrow{xo}$ Y(j) for all i,j. However, it is relatively simple to show that the implementation satisfies the aggressive conditions for SC shown in Figure 4 (i.e., the implementation directly maintains the required order between the first and last operations for any of the multiprocessor dependence chains).

A number of other aggressive implementations have been proposed for SC. Scheurich proposes a similar idea to lazy caching in his thesis, but his description of the implementation is very informal [Sch89]. Landin et al. [LHH91] propose an implementation of SC based on "race-free networks." Such networks maintain certain transactions in order, and this inherent order allows a processor to continue issuing operations as soon as its previous write reaches the appropriate "root" in the network (as opposed to waiting until the write reaches its destination caches). This is similar to the lazy caching technique since the write reaching the root in a race-free network is analogous to it being placed in the input queues of other processors in a bus-based system. Collier proposes a similar implementation for ring-based systems where one node in the ring acts as the "root" that serializes all writes [Col92]. Adve and Hill [AH90a] also propose an implementation of SC for cache-based systems that in certain cases allows a processor that has issued a write to proceed as soon as the write is serialized (as opposed to delaying the processor for the write to take affect in all cache copies). While none of the above techniques satisfy the conservative conditions for SC, they all satisfy the aggressive conditions.

Above, we discussed aggressive implementations in the context of SC. Similar optimizations are possible for more relaxed models. Landin et al. extend their observations for SC on race-free networks to aggressive implementations of PC on such networks. Again, this implementation satisfies the aggressive conditions for PC (provided in Appendix B).[19] Dubois et al.'s delayed consistency implementation of release consistency [DWB+91] delays doing the invalidations at a destination cache until the destination processor reaches an acquire as opposed to performing the invalidations before the release on the source processor. While this does not directly satisfy the original conditions for RC [GLL+90], it can be easily shown to satisfy the aggressive conditions for RC presented in Appendix B. Adve and Hill's implementations for the data-race-free-0 [AH90b] and data-race-free-1 [AH93] programmer-centric models also use an aggressive form of the multiprocessor dependence chain condition to provide more aggressive implementations than allowed by WO and RCsc. Zucker [Zuc92] makes a similar observation in the context of software cache coherence, and finally, similar observations have led to more efficient software distributed shared memory implementations [BZ91, KCZ92].

The implementations we have described above exploit the fact that maintaining the order among the intermediate operations in a multiprocessor dependence chain is not necessary for correctness. Our framework for specifying the system requirements for a model exposes this optimization. This leads to efficient implementations similar to those described above and possibly allows one to explore more efficient implementations than those already proposed. Furthermore, verifying that an aggressive implementation satisfies these conditions is relatively simple since it involves showing that each condition is satisfied directly (of course, being more conservative than the conditions require trivially satisfies them). In contrast, when the conditions are given in a conservative form (e.g., as in Figure 2), coming up with an aggressive implementation and verifying its correctness requires more involved reasoning because the orders specified by the conservative conditions are not directly upheld by the implementation and doing the verification involves proving that the executions on such an implementation are the same as if the conservative conditions were upheld.

3.1.3 Atomic Read-Modify-Writes

The read-modify-write condition requires such operations to behave atomically with respect to other writes to the same location. This is typically achieved either by atomically modifying the location at the memory or by obtaining exclusive ownership of the location and doing the atomic modification in the cache of the issuing processor.

3.1.4 Termination Condition

The termination condition imposes restrictions of the following form: for a certain write W, given the initial $W_{init}(i)$ sub-operation of the write is in the execution, other sub-operations of that write (i.e., W(1) ... W(n)) must also appear in the execution; along with our assumption that only a finite number of sub-operations are ordered before any given sub-operation in the execution order, this implies that the remaining write sub-operations can not be delayed indefinitely. This condition is required to ensure other processors will eventually see the effect of a write. The termination condition is especially relevant for systems that replicate data (e.g., through caching). Cache-based systems that use hardware protocols for coherence usually satisfy the termination condition for all sub-operations through the coherence protocol. Other systems such as Munin [CBZ91] and Midway [BZ91] which use software consistency management need to explicitly ensure that the required sub-operations are executed.

Note here that the termination condition interacts in a subtle way with the uniprocessor correctness condition (Condition 1). The latter condition imposes restrictions that require the R(i) or $W_{init}(i)$ sub-operation of a given operation to appear in the execution order. Given that the uniprocessor correctness condition results in the initial sub-operation of the write in the execution order, the termination condition then specifies the other sub-operations of the write that are required to execute. Thus, the uniprocessor correctness condition imposes restrictions on which operations should be executed by the processor environment while the termination condition is an analogous condition for the memory system. In particular, the termination condition prohibits implementations that might deadlock, and prohibits the processor from always giving priority to certain types of sub-operations (such as reads) over others (such as writes) for unbounded time.

[19]They discuss two implementations of PC, one of which does not obey the coherence condition. The implementation without coherence does not satisfy the semantics of PC as we have been using it and as defined in [GLL+90, GGH93].

3.2 Compiler Issues

The previous section described the impact of the specified system requirements on the implementation of the architecture. These system requirements have an impact on the compiler as well. Many useful compiler optimizations involve reordering or eliminating memory operations (e.g., register allocation, code motion, loop interchange) and the optimizations have to be done according to the conditions specified for a memory model. This section discusses the implications of our specification on the type of optimizations that a compiler is allowed to do given a specific memory model. Note that when applying these conditions, the compiler needs to consider all possible executions for all possible sets of input data for a given program.

Many of the aggressive aspects of our specification methodology are relevant at the architectural level only and are not likely to be exploited by the compiler. For example, compilers usually manipulate only the program order within each process' code and manipulate memory accesses at the granularity of operations (as opposed to sub-operations). Therefore, the aggressive form of the multiprocessor dependence chain condition is not likely to be exploited. For a similar reason, the coherence condition may not be relevant to the compiler. (An exception is a system that implements software-based cache coherence using instructions such as *cache invalidate* (as in the RP3 [PBG+85]).) Below, we discuss the impact of the other conditions that are relevant to the compiler. We first discuss how to conceptually reason about optimizations such as register allocation. Next we describe how to determine the set of safe optimizations that a compiler can do based on a model's constraints.

3.2.1 Reasoning about Compiler Optimizations

For a set of compiler optimizations to obey a given memory model, it is sufficient that for any execution E of the optimized code, if E obeys the requirements of the memory model, then the instructions and sub-operations of E (along with the values read and written by them) form an execution of the unoptimized code that obeys the model requirements. However, this would preclude the use of optimizations such as register allocation and common sub-expression elimination that eliminate or substitute memory sub-operations and/or instructions. To reason about such optimizations, we consider executions (that obey the memory model) of the optimized code, but add back the deleted memory sub-operations and replace new instructions by original instructions of the unoptimized code. The compiler optimizations are considered correct as long as the instructions and sub-operations of the augmented execution form an execution of the unoptimized code that obeys the system specification for a model. As an example, we illustrate below how an execution can be conceptually augmented to reason about the register allocation optimization alone (i.e., no code motion, etc.). The concepts presented should be easily extendible to other optimizations such as common sub-expression elimination or combinations of such optimizations that result in the code motion or elimination of memory sub-operations (or instructions).

Consider a memory location that is allocated to a register. Below, we briefly consider the scenarios that arise due to register allocation. Typically, register allocation involves transforming a memory read to a register read (*memory-read to register-read*) or a memory write to a register write of the same value (*memory-write to register-write*). In cases where register allocation begins with a read, the memory read is still performed and the value read is written to the register (*memory-read plus register-write*). Finally, in some cases, the value of the register may be written back to its corresponding memory location (*register to memory write-back*).

Below, we consider how the memory sub-operations that are eliminated by register allocation may be added back to the execution (that obeys the assumed memory model) of the optimized code. We will refer to the W_{init} sub-operation of a write as the *initial write sub-operation* and the remaining n sub-operations as the *per-processor write sub-operations*. The transformation of the optimized code execution described below is done for one processor at a time until all processors are covered. For each processor, we traverse its instruction instances in program order. For a *register read* on P_i that arises from the memory-read to register-read case discussed above, we replace the register read with a read sub-operation (R(i)) to the corresponding memory location that returns the same value as the register read. To place this sub-operation in the execution order, we need to identify the last memory write, or the last memory read that corresponds to the memory-read plus register-write case above (whichever is later in program order), that is to the same location and before the added read in program order. The added read sub-operation is then placed immediately after the chosen R(i) or W_{init}(i) sub-operation. A *register write* on P_i that arises from the memory-write to register-write case is replaced with an initial write sub-operation (W_{init}(i)) that writes the same value to the corresponding memory location. The added W_{init}(i) is placed in the execution order immediately after the read or W_{init}(i) sub-operation of the last read or write to the same location (whichever is later) that is before the added write in program order. Finally, we consider a memory write W on P_i that arises from a *register to memory write-back.* We first remove the $W_{init}(i)$ sub-operation corresponding to this write from the execution (since one is already added where the register was written to). We then identify all the previous (in program oder) register writes (that have been transformed to memory writes) since the last write-back; for each such write W', we add the per-processor write sub-operations (i.e., W'(1) ... W'(n)) to the execution order with each W'(j) being placed immediately before the W(j) sub-operation corresponding to the current write-back (the execution order among the added write sub-operations to the same memory copy should correspond to the program order among those sub-operations).

Given the above augmented execution, the register allocation optimizations are considered safe if the instructions and sub-operations of the augmented execution form an execution of the unoptimized program and if this execution obeys the system requirements for a given model. This methodology can be extended to reason about the correctness of other combinations of optimizations.

3.2.2 Safe Compiler Optimizations

We now explore the optimizations that the compiler can perform based on the constraints imposed by a given memory model. To satisfy the notion of uniprocessor data dependence upheld by the value, initiation, and uniprocessor dependence conditions, the compiler needs to ensure that if a read returns the value of its own processor's write, then this write should be the last write before the read by the program order of the source program. In the absence of optimizations such as register allocation, this is easily achieved if the compiler does not reorder a write followed (in program order) by a read to the same location. With register allocation, the condition is maintained as long as within the interval in which the location is register allocated, the program ordering of the conflicting register operations in the compiled program is the same as the program ordering of the corresponding memory operations in the original source program that are replaced by register allocation. This ensures that if a read returns the value of its own processor's write, then it will always return the value of the last conflicting write before it by program order as in the original source program. In addition, the uniprocessor dependence condition requires that the program order among all conflicting accesses be maintained.[20] Thus, the notion of uniprocessor data dependence can be maintained similar to the way a uniprocessor compiler maintains it.

To enforce the multiprocessor dependence chain, it is sufficient if the compiler disallows reordering among memory operations related by the $\xrightarrow{po}$ arcs (e.g., $\xrightarrow{spo}$) that occur in any of the specified chains (refer to Appendix A for implications of the reach relation). We use $\xrightarrow{spo}$ to generically refer to the variety of $\xrightarrow{po}$ arcs that make up such chains. One of the requirements on the compiler is to preserve the labels (and fences, e.g. in PSO) in a program in order to ensure that the $\xrightarrow{spo}$ order remains visible by the underlying system. Again, while we can satisfy the multiprocessor dependence chain aggressively in hardware without maintaining order among the intermediate operations, we expect this type of flexibility will not be exploitable by the compiler. Below, we briefly discuss the restrictions placed on register allocation arising from the multiprocessor dependence chains. The conservative approach is to (i) disallow replacing a memory read R with a register read if the value of the register is set prior to the previous operation ordered before R by $\xrightarrow{spo}$, and (ii) disallow replacing a memory write W with a register write if the write-back operation corresponding to W (or a later write in program order) does not appear in program order before the next operation ordered after W by $\xrightarrow{spo}$.[21] Furthermore, for case (i), register allocation of R should be disallowed if the value of the register is set by another read R' and the label for R is more conservative than the label for R', and for case (ii), register allocation of W should be disallowed

[20]The interaction between the value, initiation, uniprocessor dependence, and coherence conditions discussed in Section 3.1.1 also applies to compiler optimizations. For example, if the uniprocessor dependence condition for a model implies $\xrightarrow{sxo}$ between W $\xrightarrow{po}$ R to the same location, and if the write is covered by the coherence condition, then replacing the memory operations by register reads and writes may lead to incorrect executions. The reason is that register allocating W and R can result in the sub-operation of R to execute in its memory copy before the sub-operation of W in that memory copy. In such cases, more analysis is required to determine whether register allocation is safe. However, note that the aggressive specifications for models presented in this paper all exclude the W $\xrightarrow{po}$ R case from their coherence conditions.

[21]Register allocation can be done more aggressively than described above. For example, given X $\xrightarrow{spo}+$ R, R may be replaced by a register read if the value that it reads belongs to a write that is also register allocated and whose value is not written back to memory until after X in program order. Similarly, as mentioned above, given W $\xrightarrow{spo}+$ Y, W may be replaced by a register write as long as there is a write back to memory (of the value written by W or by a later write to the same location) before Y in program order. However, note that the value that remains in the register may still be used to replace read references after Y as long as the other conditions allow this.

```
P1                      P2
A = 1;                  B = 1;
while (B == 0);         while (A == 0);
```

Figure 8: Code segment example for compiler optimizations.

if the label of W is more conservative than the label of the write-back memory operation (e.g., W is a release while the write-back memory operation is an ordinary operation). In addition, given $X \xrightarrow{spo}+ Y \xrightarrow{spo}+ Z$, if register allocating Y eliminates $X \xrightarrow{spo}+ Z$, we conservatively disallow it.[22]

Finally, the compiler needs to ensure the termination condition; i.e., for certain writes, the compiler must ensure that the write operation occurs in all executions of the compiled code. To ensure this, such a memory write should not be completely eliminated by optimizations such as register allocation. For example, if a memory write is converted to a register write, we have to guarantee that the register value will eventually get written to memory. A related issue is that reads that may possibly be issued an infinite number of times should not be eliminated by optimizations such as register allocation either. For example, register allocating a read within a loop may result in the old value for the location to be used in every iteration of the loop and new writes to the same location that appear in the execution order would not affect the value returned; this would violate Condition 2(a) (in the transformed version of the program where we add back the register-allocated memory operations as discussed in the previous section) because it will appear as if there are an infinite number of read sub-operations ordered before the new write sub-operation to the same memory copy.

The above explained how the compiler can satisfy the model specific requirements. The uniprocessor correctness condition which is implicitly assumed for all models (Condition 1) imposes additional constraints on the compiler. The requirement is that the memory sub-operations that execute for each process should correspond to a "correct" uniprocessor execution of the process' program given the reads return the same values as the multiprocessor execution. The intuitive notion of correctness for a uniprocessor compiler is for most part sufficient for maintaining this condition. However, there are certain aspects that are specific to multiprocessors. For example, an initial write sub-operation W_{init} that would occur in the correct uniprocessor execution should also occur in the multiprocessor execution. Another case is if an operation A is followed by an operation B where B can execute for an infinite number of times, then the compiler should not reorder B before A since this can cause A to never execute. Similarly, if a write operation D is preceded by an operation C where C can execute for an infinite number of times, then D should not be reordered before C. This is because D may execute in the "optimized" execution while it would not have executed in the corresponding "unoptimized" execution. Appendix D discusses how this constraint may be relaxed.

Below, we present different scenarios that can result in violations of the termination, value, and uniprocessor correctness conditions. Refer to the code segment in Figure 8. Assume a model that requires the termination condition to hold for P1's write of A, and for P2's write of B. The uniprocessor correctness, value, and termination conditions together guarantee that any possible executions of this code should result in both loops to terminate. However, any of the following optimizations may violate this property: (1) if the while loop on each processor is moved above the write on that processor, (2) if the writes of A and B are done as register writes and the new values are not written back to memory, or (3) if location B is register allocated in P1 or location A is register allocated in P2. The first optimization is disallowed by the uniprocessor correctness condition, the second by the termination condition, and the third by Condition 2(a) (as discussed above) because only a finite number of sub-operations (reads in this case) may precede any given sub-operation (the write in this case) in the execution order.

[22]For example, consider the aggressive conditions for RCsc given in Appendix B. Assume $Rc \xrightarrow{po} Rc\_acq \xrightarrow{po} W$. Therefore, $Rc \xrightarrow{spo} W$ holds. However, if we register allocate the acquire reference, the $\xrightarrow{spo}$ may no longer hold between the read and write. To be more aggressive, a compiler may use implementation-dependent information to determine whether register allocating the acquire results in possible reordering of the other two operations.

4 Discussion and Related Work

This section compares our abstraction and framework with already existing approaches. The second part discusses implications of some of the design decisions for our framework.

4.1 Comparison to Other Approaches

Below we discuss various approaches that have been proposed for specifying system requirements for memory models. We distinguish between the system abstraction used by each approach and the methodology used to specify the ordering constraints. For the system abstraction, we are interested in whether it is flexible enough for expressing the desirable optimizations. For the specification methodology, we compare the approaches based on aggressiveness of the constraints.

Dubois et al. [DSB86, SD87] present an abstraction based on the various stages that a memory request goes through and use this abstraction to present specifications for both sequential consistency and weak ordering. As discussed in Section 2, the notion of "perform with respect to a processor" in this abstraction models the effects of replication and the non-atomicity of writes in real systems. One of the problems with this abstraction is that the definition of "perform" is based on real time. Another shortcoming of the abstraction is that it does not seem to be powerful enough to capture the optimization where the processor is allowed to read the value of its own write before the write takes effect in any memory copy (as discussed in Sections 2 and 3.1.1).[23]

The abstractions proposed by Collier [Col92], Sindhu et al. [SFC91], and Gibbons et al. [GMG91, GM92] were also referred to in Section 2. Collier's abstraction is formal and captures replication of data and the non-atomicity of writes. Yet it has the same shortcoming as Dubois et al.'s abstraction in its inability to capture the optimization where a read returns the value of its own processor's write before the write is serialized. Note that our abstraction subsumes Collier's abstraction. In particular, our abstraction degenerates to Collier's abstraction if we remove the W_{init} sub-operation and require W $\xrightarrow{po}$ R to imply W(i) $\xrightarrow{xo}$ R(i) when both operations are to the same location. However, these two notions are important for properly capturing the mentioned optimization. Considering other abstractions, Sindhu et al.'s abstraction [SFC91] was designed to describe TSO and PSO and therefore handles the optimization described above, but is not flexible enough to capture the non-atomicity of writes. Another way of abstracting the system is to represent it in terms of execution histories [HW90] (see [AF92] for another example where this abstraction is used). Effectively, a history represents one processor's view of all memory operations or a combined view of different processors. This type of abstraction is in essence similar to Collier's abstraction and shares the same advantages and disadvantages. Gibbons et al.'s abstraction [GMG91, GM92] is the only one that captures replication and non-atomicity of writes and is flexible enough for specifying models such as TSO. However, while the I/O automaton model they use leads to precise specifications, the specifications typically model the system at too detailed a level[24], making it complex to reason with and difficult to apply to system designs with substantially different assumptions.

We now consider the methods for specifying system requirements used with the above abstractions. Dubois et al.'s specification style [DSB86, SD87] leads to unnecessary constraints on memory ordering since it constrains the execution order among accesses to different locations in a similar way to the conservative conditions for SC in Figure 2. This disallows the type of aggressive implementations discussed in Section 3.1.2. This same limitation exists with the specification by Sindhu et al. [SFC91] for TSO and PSO and the specification by Gibbons et al. [GMG91, GM92] for release consistency. Collier [Col92] does observe that two sets of conditions are indistinguishable if they maintain the same order among conflicting accesses, yet his methodology for specifying conditions constrains order among non-conflicting operations just like the other schemes. Therefore, none of the above methodologies expose the optimizations that become possible when only the order among conflicting operations is constrained.

Adve and Hill's specification of sufficient conditions for satisfying DRF1 [AH92] is one of the few specifications that presents ordering restrictions among conflicting memory operations only. However, parts of these conditions are too general to be easily convertible to an implementation. While they provide a second set of conditions that

[23]See [GGH93] for a discussion on this limitation of Dubois et al.'s framework and an extension to it that allows this optimization to be expressed.

[24]Gibbons et al.'s specification [GMG91, GM92] involves more events than we use in our specification and inherently depends on states and state transitions.

translates more easily to an implementation, this latter set of conditions are not as aggressive and restrict orders among operations to different locations. Finally, because their specification is based on Collier's abstraction, their approach does not easily lend itself to specifying models that use the optimization of allowing a processor to read its own write before the write is serialized.

In summary, the system abstraction proposed in this paper is more general than most previously proposed abstractions. Furthermore, our specification methodology exposes more optimizations and is easier to translate into aggressive implementations than previous methods.

4.2 Choice of Abstraction and Framework

In designing our specification technique, our primary goals have been to provide a framework that covers both the architecture and compiler requirements, is applicable to a wide range of designs, and exposes as many optimizations as possible without violating the semantics of a memory model. Our specification framework could conceivably be different if we chose a different set of goals. For example, with the general nature of our framework, the designer may have to do some extra work to relate our conditions to a specific implementation. Had we focused on a specific class of implementations, it may have been possible to come up with an abstraction and a set of conditions that more closely match such designs. Similarly, our methodology of only restricting the order among conflicting operations is beneficial mainly at the architectural level. This complexity would not be very useful if we wanted to only specify requirements for the compiler. And in fact, such complexity is undesirable if the specification is to only be used by programmers to determine the set of possible executions under a model.[25] However, we feel the benefit of providing a uniform framework, that covers both the architecture and the compiler and applies to a wide range of implementations, outweighs any of its shortcomings.

The abstraction and framework presented in this paper is general and can capture a variety of memory models and specifications. We provide the specification for several of the hardware-centric models in Appendix B. We have also used our framework for specifying the system requirements for programmer-centric models such as PLpc [AGG$^+$93]. Given the generality of our framework, it would be interesting to use it to specify the system requirements for other models that we have not discussed in this paper. The fact that a uniform framework may be used for specifying different models can greatly simplify the task of comparing the system implications across the various models.

5 Concluding Remarks

This paper described a framework for specifying sufficient system requirements to satisfy the semantics of various memory consistency models. Our specification methodology has two major advantages compared to previous approaches. First, it exposes many aggressive implementation optimizations that do not violate the semantics of a given model. The key observation that allows us to do this is that constraining the execution order only among conflicting operations is sufficient for specifying the semantics for any model. The second advantage of our methodology is that converting our specifications into either conservative or aggressive implementations and conversely verifying that an implementation satisfies the specification are relatively straightforward. These characteristics aid the designer in implementing a memory model correctly and efficiently across a wide range of system designs.

References

[ABM89] Y. Afek, G. Brown, and M. Merritt. A lazy cache algorithm. In *Symposium on Parallel Algorithms and Architectures*, pages 209–222, June 1989.

[Adv93] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Technical Report #1198, University of Wisconsin - Madison, December 1993.

[25]Our abstraction is meant to be used by system designers and we don't expect the programmer to reason with this specification. We strongly believe programmers should reason with the high-level abstraction presented by programmer-centric models.

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 679–690, May 1992.

[AGG+93] Sarita V. Adve, Kourosh Gharachorloo, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Sufficient system requirements for supporting the PLpc memory model. Technical Report #1200, Computer Sciences, University of Wisconsin - Madison, December 1993. Also available as Stanford University Technical Report CSL-TR-93-595.

[AH90a] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I: 47–50, August 1990.

[AH90b] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[AH92] Sarita V. Adve and Mark D. Hill. Sufficient conditions for implementing the data-race-free-1 memory model. Technical Report #1107, Computer Sciences, University of Wisconsin - Madison, September 1992.

[AH93] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BZ91] B. Bershad and M. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Col92] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, Inc., 1992.

[DSB86] 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.

[DWB+91] M. Dubois, J.-C. Wang, L.A. Barrosso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proceedings of Supercomputing '91*, pages 197–206, 1991.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GAG+92] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[GGH92] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceeding of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[GGH93] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to "Memory consistency and event ordering in scalable shared-memory multiprocessors". Technical Report CSL-TR-93-568, Stanford University, April 1993.

[GLL$^+$90] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GM92] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *Symposium on Parallel Algorithms and Architectures*, pages 158–168, June 1992.

[GMG91] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[Goo91] James R. Goodman. Cache consistency and sequential consistency. Technical Report Computer Sciences #1006, University of Wisconsin, Madison, February 1991.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LHH91] A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 27–30, May 1991.

[LLG$^+$90] 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*, pages 148–159, May 1990.

[MPC89] Samuel Midkiff, David Padua, and Ron Cytron. Compiling programs with user parallelism. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[PBG$^+$85] 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.

[Sch89] Christoph Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989.

[SD87] 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.

[SFC91] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report (PARC) CSL-91-11, Xerox Corporation, Palo Alto Research Center, December 1991.

[SS88] 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.

[SUN91] *The SPARC Architecture Manual*. Sun Microsystems Inc., January 1991. No. 800-199-12, Version 8.

[ZB92] Richard N. Zucker and Jean-Loup Baer. A performance study of memory consistency models. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–12, May 1992.

[Zuc92] Richard N. Zucker. *Relaxed Consistency and Synchronization in Parallel Processors*. PhD thesis, University of Washington, December 1992. Also available as Technical Report No. 92-12-05.

Appendix A: Reach Relation

This appendix defines the reach ($\xrightarrow{rch}$) relation for the models discussed in this paper. The reach relation is used in the conditions for the WO, RCsc, and RCpc models as discussed in Section 2.3.2. Part of the reach relation is also used in Appendix D to relax the uniprocessor correctness condition. This relation is a straightforward adaptation of the $\xrightarrow{rch}$ relation developed for the PLpc model and the formalism given below is taken from [AGG+93]. The appendix concludes with a description of the implications of the $\xrightarrow{rch}$ relation on implementations, and a note on how this relation could potentially be further relaxed.

Similar to the uniprocessor correctness condition, the reach relation depends on all program instructions as opposed to only instructions reading and writing shared locations. Thus, when determining the reach relation for an execution, we need to consider the set of instruction instances, I, of the execution. To formalize the reach relation, we first present an abstraction for the program instruction statements. We classify instructions into three types: *computation*, *memory*, and *control*. Computation instructions read a set of registers (*register read set*) and map the read values into new values that are written to another (possibly same) set of registers (*register write set*). Memory instructions are used to read and write memory locations (both private and shared). A memory read instruction reads the address to be read from a register, reads the specified memory location, and writes the return value into a register. A memory write instruction reads the address and value to be written from registers and writes the value into the specified memory location. A memory read-modify-write instruction is both a memory read and a memory write instruction that reads and writes the same location. Therefore, for the read instruction, the register read set comprises of the address register and the register write set is the destination register. For the write instruction, the register read set comprises of the address and value registers and there is no register write set. Finally, control instructions (e.g., branch instructions) change the control flow by appropriately adjusting the program counter. The register read set for a control instruction is a set of registers whose values determine the change in the control flow. If an instance of a control instruction results in the program counter only being incremented, then we say the instruction instance preserves program control flow; if the program counter changes in any other way, we say the instruction instance results in changing the program control flow. Note that the above description of instructions is merely an abstraction and can be adapted to most architectures and languages.

We now define the notion of local data dependence ($\xrightarrow{dd}$) and control dependence ($\xrightarrow{cd}$) that will be used to develop the $\xrightarrow{rch}$ relation. For two instruction instances A and B in an execution, A $\xrightarrow{dd}$ B if B reads a value that A writes into its register write set or a private memory location of its processor. (The interaction due to shared-memory data dependence is handled later.) Our notion of control dependence is borrowed from Ferrante et al. [FOW87]. Control dependence is defined in terms of a control flow graph and dominators [ASU86]. Let Prog be the program under consideration, and let E be an execution of program Prog. Consider the control flow graph of any process in program Prog, with the final node in the graph denoted by EXIT. Let C' and D' be two instructions in the control flow graph. C' is *post-dominated* by D' if D' occurs on every path in the control flow graph from C' to EXIT. Let A and B be two instruction instances of processor P in execution E and let A' and B' be the instructions corresponding to A and B in the program text. Instruction instance B is *control dependent* on instruction instance A if the following conditions hold: (i) A $\xrightarrow{po}$ B in execution E, *and* (ii) A' is not post-dominated by B', *and* (iii) there is a path between A' and B' in the control flow graph of processor P such that all the nodes on this path (excluding A',B') are post-dominated by B'. Note that if A $\xrightarrow{cd}$ B, then A' is a control instruction.

To allow for possibly non-terminating executions, we need to augment the control flow graph and the resulting $\xrightarrow{cd}$ relation with additional arcs. Informally, consider a loop in the program that does not terminate in some SC execution. Then for any instruction instance *i* that is program ordered after an instance of the loop, we require $\xrightarrow{cd}+$ to order *i* after instances of the control instructions of the loop that change the program flow for this loop and cause infinite execution. More formally, let C' be any control instruction that could be executed infinite times in some SC execution E. Suppose an infinite number of successive instances of C' change the control flow of the program in E. Add an auxiliary edge from every such instruction C' to the EXIT node. This ensures that any such control instruction C' is not post-dominated by any of the instructions that follow it in the control flow graph, and so instances of C' are ordered before all instances of all subsequent instructions by $\xrightarrow{cd}+$. The modification described above is not necessary if all SC executions of the program will terminate, or if there are no memory operations that are ordered after possibly non-terminating loops in the control flow graph.

We next use $\xrightarrow{cd}$ and $\xrightarrow{dd}$ to define two relations, the uniprocessor reach dependence ($\xrightarrow{Udep}$) and the multiprocessor reach dependence ($\xrightarrow{Mdep}$) below, and then define the $\xrightarrow{rch}$ relation in terms of these two new relations. A discussion of the different relations and the definition of the $\xrightarrow{rpo}$ relation used in the definition of $\xrightarrow{Mdep}$ follow after the formal definitions.

Definition A.1: Uniprocessor Reach Dependence
Let X and Y be instruction instances in an execution E of program Prog. X $\xrightarrow{Udep}$ Y in E iff X $\xrightarrow{po}$ Y, and either
(a) X $\xrightarrow{cd}$ Y in E, or
(b) X $\xrightarrow{dd}$ Y in E, or
(c) X and Y occur in another possible SC execution E' of the program Prog, X $\xrightarrow{cd}$+ Z $\xrightarrow{dd}$ Y in E', and Z does not occur in E.

Definition A.2: Multiprocessor Reach Dependence
Let X and Y be instruction instances in an execution E of program Prog. Let Y be an instruction instance that accesses shared-memory. X $\xrightarrow{Mdep}$ Y in E iff any of the following are true.
(a) X $\xrightarrow{po}$ Y in E and X and Y occur in another possible SC execution E' of Prog, where X $\xrightarrow{Udep}$+ Z $\xrightarrow{rpo}$ Y in E', Z is an instance of a shared-memory instruction, for any A $\xrightarrow{dd}$ B constituting the $\xrightarrow{Udep}$+ path from X to Z in E', B also occurs in E, and either Z does not occur in E, or Z occurs in E but is to a different address in E and E', or Z is a write to the same address in E and E' but writes a different value in E and E'.
(b) X $\xrightarrow{po}$ Y in E and X $\xrightarrow{Udep}$ Z $\xrightarrow{rpo}$ Y in E.
(c) X $\xrightarrow{po}$ Y in E and X $\xrightarrow{Mdep}$ Z $\xrightarrow{rpo}$ Y in E.
(d) X $\xrightarrow{po}$ Y in E or X is the same as Y. Further, X generates a competing read R (in RCsc,RCpc) or synchronization read R (in WO) within a loop and Y generates an operation O (different from R) such that R $\xrightarrow{rpo}$+ Y.

Definition A.3: Reach' Relation
Given an execution E and instruction instances X and Y in E (where X may or may not be the same as Y), X $\xrightarrow{rch'}$ Y in E iff X and Y are instances of memory instructions, X generates a shared-memory read, Y generates a shared-memory write, and X $\{\xrightarrow{Udep} \mid \xrightarrow{Mdep}\}$+ Y in E. For two different memory operations, X' and Y', from instruction instances X and Y respectively, X' $\xrightarrow{rch'}$ Y' iff X' is a read, Y' is a write, X' $\xrightarrow{po}$ Y' and X $\xrightarrow{rch'}$ Y.

Definition A.4: Reach Relation
Given an execution E and instruction instances X and Y in E, X $\xrightarrow{rch}$ Y in E iff X $\xrightarrow{rch'}$ Y and X generates a memory read that reads the value of another processor's write. (The $\xrightarrow{rch}$ relation among memory operations is defined in an analogous manner to $\xrightarrow{rch'}$.)

The reach relation is a transitive closure of the uniprocessor reach dependence ($\xrightarrow{Udep}$) and the multiprocessor reach dependence ($\xrightarrow{Mdep}$) relations. The uniprocessor component corresponds to uniprocessor data and control dependence, while the multiprocessor component corresponds to dependences that are present due to the memory consistency model. The components that make up $\xrightarrow{Udep}$ and $\xrightarrow{Mdep}$ are defined for a given execution E. Both relations also require considering other sequentially consistent executions of the program, and determining if an instruction in one execution occurs in the other execution.[26] For an instruction instance from one execution to occur in another, we do not require that locations accessed or the values read and written by the corresponding instruction instances in the two executions be the same; we are only concerned with whether the specific instances appear in the execution. In the absence of constructs such as loops and recursion, it is straightforward to determine if an instance that appears in one execution also appears in another. With loops and recursion, care has to be taken to match consistent pairs of instruction instances. Instruction instances between two executions are matched consistently if the set of instances that are considered to appear in both executions have the same program order

[26]It may be simpler to be conservative and consider all possible executions of the program and not just SC executions for E' in the definitions of $\xrightarrow{Udep}$ and $\xrightarrow{Mdep}$.

```
(s1) r1 = A;
(s2) if (r1 == 1) {(s3) r2 = B;}
(s4) C = r2;
```

(a)

```
(s1) r1 = D;
(s2) if (r1 == 1) {(s3) r2 = FLAG; (acq) }
(s4) E = 1;
```

(b)

Figure 9: Example code to illustrate the reach condition.

relation between them in both executions, and are the maximal such sets. (A set S with property P is a maximal set satisfying property P if there is no other set that is a superset of S and also satisfies property P.)

To illustrate the above concepts, consider the example in Figure 9(a). The variables in uppercase are shared memory locations; the symbols in lowercase refer to registers. The symbols in parentheses are labels for the instruction statements. Assume an execution E where the read of A returns the value of 0. Thus, r1 is 0 and the read of B and assignment to r2 do not occur in this execution E. Therefore, in E, there is no instruction instance ordered before (s4) by $\xrightarrow{cd}$ or by $\xrightarrow{dd}$. Thus, parts (a) and (b) of the definition of $\xrightarrow{Udep}$ do not order any instruction instance before (s4) by $\xrightarrow{Udep}$. However, if we consider a possible SC execution E' where the return value of A is 1 (assume it is the value of another processor's write), then we have (s2) $\xrightarrow{cd}$ (s3) $\xrightarrow{dd}$ (s4) in E', (s3) does not occur in E, and so (s2) $\xrightarrow{Udep}$ (s4) in E by part (c) of the definition of $\xrightarrow{Udep}$. Further, (s1) $\xrightarrow{dd}$ (s2) in E and so (s1) $\xrightarrow{Udep}$ + (s4) in E and so (s1) $\xrightarrow{rch}$ (s4) in E.

The $\xrightarrow{Mdep}$ relation also requires specifying a given memory model. There is a circularity however since the model specifications that use the $\xrightarrow{rch}$ relation may depend on the $\xrightarrow{Mdep}$ relation (through $\xrightarrow{rch}$), and yet, we say the $\xrightarrow{Mdep}$ relation depends on the model specification. For the purposes of defining $\xrightarrow{Mdep}$ (and $\xrightarrow{rch}$) for a model, we consider the aggressive specification of the model with the $\xrightarrow{rch}$ relation removed. The parts of the specification that we are interested in are the uniprocessor dependence part of the $\xrightarrow{sxo}$ condition, the initiation condition, and the $\xrightarrow{spo}$ relation with the $\xrightarrow{rch}$ relation removed (called $\xrightarrow{spo}$- below). [27]

Definition A.5: $\xrightarrow{rpo}$ Relation
Let X and Y be instances of shared-memory instructions in an execution E, and let X' and Y' be the memory operations corresponding to X and Y respectively in E. X $\xrightarrow{rpo}$ Y in E iff either
(a) X' $\xrightarrow{spo}$- Y' in E, or
(b) X' $\xrightarrow{po}$ Y' in E and X' $\xrightarrow{po}$ Y' is in the uniprocessor dependence part of the $\xrightarrow{sxo}$ condition, or
(c) X'=W $\xrightarrow{po}$ Y'=R in E and the initiation condition requires that $W_{init}(i)$ $\xrightarrow{xo}$ R(i) in E.

For example, for RCpc (Figure 5), the $\xrightarrow{rpo}$ relation is as follows. Let X and Y be instruction instances and let X' and Y' be memory operations corresponding to X and Y respectively. X $\xrightarrow{rpo}$ Y if X' $\xrightarrow{spo'}$ Y', or if X',Y' conflict and X' $\xrightarrow{po}$ Y'. To illustrate the use of $\xrightarrow{rpo}$ in $\xrightarrow{Mdep}$ (definition A.2 above), consider the example in Figure 9(b) under the RCpc model. Assume an execution E where the return value for the read of D is 0. Therefore, (s3) does not occur in E. Consider a possible SC execution E' where the read of D returns 1 (assume the value is from another processor's write), resulting in (s3) to be executed. Assume all memory accesses are to shared locations. Under RCpc, R_acq $\xrightarrow{po}$ W constitutes R_acq $\xrightarrow{rpo}$ W. Therefore, in E', (s3) $\xrightarrow{rpo}$ (s4). Since we have (s2) $\xrightarrow{cd}$ (s3) $\xrightarrow{rpo}$ (s4) in E' and (s3) does not occur in E, we have (s2) $\xrightarrow{Mdep}$ (s4) in E by Definition A.2(a). Further, (s1) $\xrightarrow{dd}$ (s2) in E and so (s1) $\xrightarrow{Udep}$ (s2) $\xrightarrow{Mdep}$ (s4) in E and so (s1) $\xrightarrow{rch}$ (s4) in E. Note that in this example, $\xrightarrow{Udep}$ does not hold between (s2) and (s4), and in the example on the left (Figure 9(a)), $\xrightarrow{Mdep}$ does not hold between (s2) and (s4).

We now discuss the implications of the $\xrightarrow{rch}$ relation on the hardware and software implementations of WO, RCsc, and RCpc. This relation is involved in the multiprocessor dependence chain part of the aggressive conditions

[27]We allow the relation X $\xrightarrow{spo}$ Y to be dependent on the addresses of X and Y. In this case, if the same pair of memory instructions accesses different addresses in two executions, it is possible for the pair to be ordered by $\xrightarrow{spo}$ in one execution but not in the other.

and in the $\xrightarrow{sxo'}$ part of the conservative conditions. (The $\xrightarrow{rch'}$ relation is also used later in Appendix D to relax the uniprocessor correctness condition for all models; the implications on implementations of this relaxation are discussed in appendix D.)

The use of the $\xrightarrow{rch}$ relation formalizes and extends the control condition developed for the DRF1 memory model [AH92] and so the discussion of the implications for implementations is similar to that in [AH92]. The most conservative way of satisfying the conditions that involve the $\xrightarrow{rch}$ relation is to make a processor stall on a read until the read returns its value. A more aggressive implementation can allow a processor to proceed with pending reads as long as it does not execute a write sub-operation W(j) until (a) it is resolved that W will indeed be executed (i.e., will not have to be rolled back) with the given address and value, and (b) it is known which memory operations before W by program order will be executed, which addresses such operations will access, and which values such writes will write. Thus, reads can always be executed speculatively, but writes need to wait until the control flow for the execution and the addresses and values to be accessed by different operations before W are resolved.[28] Note that the augmenting of the control flow graph discussed earlier in this appendix implies that for a loop that is not guaranteed to terminate in an SC execution and whose termination depends on the value returned by a shared-memory read R, writes following the loop in the control flow graph cannot be executed until R returns its value. The implementation described above is more aggressive than many previous specifications. Of course, we can be even more aggressive by directly following the specifications in the conservative and aggressive conditions given for the models in Appendices B and C.

The constraints on the compiler of a WO, RCsc, or RCpc system due to the $\xrightarrow{rch}$ relation are analogous to those for the hardware. Consider a shared-memory write instruction W that follows (by program order) a shared-memory read instruction R in the unoptimized source code. If an instance of R could be ordered before an instance of W by $\xrightarrow{rch}$ in any execution of the unoptimized source code on a system of the corresponding model, then the compiler cannot move W to before R in the optimized code. Also, similar to the hardware, for a loop that is not guaranteed to terminate in all SC executions and whose termination depends on the value returned by a shared-memory read R, the compiler cannot move a write that follows the loop (by program order in the unoptimized source code) to before R.

Finally note that, as discussed in Section 2.3.2, the conditions for WO, RCsc, and RCpc use the $\xrightarrow{rch}$ relation to ensure that the models appear SC to a set of well-behaved programs. We have shown that the $\xrightarrow{rch}$ relation formalized above is sufficient for this purpose [Adv93]; however, we do not know that it is necessary and believe that certain aspects such as requiring W $\xrightarrow{po}$ R as part of $\xrightarrow{rpo}$ in part (c) of definition A.2 may be overly conservative. In the future, it may be possible to relax the $\xrightarrow{rch}$ relation without violating the intended semantics of the above models. The $\xrightarrow{rch'}$ relation is also used in appendix D to relax the uniprocessor correctness condition for all the models. The above comment applies to the use of the $\xrightarrow{rch'}$ relation in that context as well.

[28]Note that sub-operations that are executed speculatively but rolled back later are not included in the final set of sub-operations for the execution.

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ W

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X $\xrightarrow{co}$ Y

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R
RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.
Termination condition holds for all sub-operations.

Figure 10: Aggressive conditions for PC.

Appendix B: Aggressive Conditions for Hardware-Centric Models

This appendix presents the aggressive conditions for PC, TSO, PSO, WO, RCsc, and RCpc based on our framework (the conditions for SC were given in Section 2). In Appendix C, we prove that these aggressive conditions are semantically equivalent to the original conditions for these models; i.e., an implementation that obeys the aggressive conditions *appears as if* it obeys the original conditions and vice versa. Consequently, for the programmer, the two sets of conditions are equivalent; however, for the system designer, our conditions expose more aggressive implementations.

In the rest of the appendices, we refer to the original conditions as the conservative conditions. These conditions are based on the following sources: PC, RCsc, and RCpc [GLL+90] (includes minor modification discussed in [GGH93]); TSO and PSO [SFC91, SUN91]; WO [Sch89]. Since some of the original conditions do not precisely define certain behavior, we make the following assumptions. For WO, we assume the following: the cache coherence condition holds for all writes and the same location can be read and written by both synchronization and data operations. For RCsc and RCpc, we assume that termination holds for all competing sub-operations. For PC, we assume that termination holds for all sub-operations. For WO, RCsc, and RCpc, we use the $\xrightarrow{rch}$ relation as defined in Appendix A and explained in Section 2.3.2 to capture the notion of "uniprocessor data and control dependences" [GLL+90]. Appendix C provides the conservative form for the PC, WO, RCsc, and RCpc (with the above assumptions) expressed in our framework.

In the following conditions, we use R and W to denote any read and write operations respectively. For WO, we use Rs and Ws to denote synchronization reads and writes. For RCsc and RCpc, we use Rc and Wc to denote competing reads and writes and Rc_acq and Wc_rel to denote acquire and release operations, respectively. Note that R and W include all operations (e.g., even competing or acquire and release operations). Read-modify-writes are denoted by RMW, and AR and AW denote the individual read and write operation of a RMW. Finally, for PSO, STBAR is the store barrier operation [SFC91, SUN91].

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ W
AW (in RMW) $\xrightarrow{po}$ R
W $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R
RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R
atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.
Termination condition holds for all sub-operations.

Figure 11: Aggressive conditions for TSO.

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ W
AW (in RMW) $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R
RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R
atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.
$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.
Termination condition holds for all sub-operations.

Figure 12: Aggressive conditions for PSO.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$:

X $\xrightarrow{spo''}$ Y if X,Y are the first and last operations in one of

RWs $\xrightarrow{po}$ RWs

RW $\xrightarrow{po}$ RWs

RWs $\xrightarrow{po}$ RW

X $\xrightarrow{spo'}$ Y if X $\xrightarrow{spo''}$ A $\{\xrightarrow{rch} \mid \xrightarrow{spo''}\}*$ Y

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo'}\}+$ Y

define $\xrightarrow{sco}$, $\xrightarrow{sco'}$:

X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

X $\xrightarrow{sco'}$ Y if X,Y are the first and last operations in R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on different processors

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R

RW $\xrightarrow{spo}$ $\{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo'}$ $\{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+$ R

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all sub-operations.

Figure 13: Aggressive conditions for WO.

define $\xrightarrow{spo}, \xrightarrow{spo'}, \xrightarrow{spo''}, \xrightarrow{spo'''}$:

X $\xrightarrow{spo'''}$ Y if X and Y are *competing* and Xc $\xrightarrow{po}$ Yc

X $\xrightarrow{spo''}$ Y if X,Y are the first and last operations in one of

RW $\xrightarrow{po}$ Wc_rel

Rc_acq $\xrightarrow{po}$ RW

X $\xrightarrow{spo'}$ Y if X $\xrightarrow{spo'''}$ A $\{\xrightarrow{rch} \mid \xrightarrow{spo''} \mid \xrightarrow{spo'''}\}*$ Y

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo''} \mid \xrightarrow{spo'''}\}+$ Y

define $\xrightarrow{sco}, \xrightarrow{sco'}$:

X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

X $\xrightarrow{sco'}$ Y if X,Y are the first and last operations in R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on different processors

define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R

RW $\xrightarrow{spo}$ $\{$(A $\xrightarrow{sco}$ B $\xrightarrow{spo}$) $\mid$ (A $\xrightarrow{sco'}$ B $\xrightarrow{spo'}$)$\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo'}$ $\{$(A $\xrightarrow{sco}$ B $\xrightarrow{spo}$) $\mid$ (A $\xrightarrow{sco'}$ B $\xrightarrow{spo'}$)$\}+$ R

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all *competing* sub-operations.

Figure 14: Aggressive conditions for RCsc.

define $\xrightarrow{spo}, \xrightarrow{spo'}$**:**

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ RWc

Wc $\xrightarrow{po}$ Wc

RW $\xrightarrow{po}$ Wc_rel

Rc_acq $\xrightarrow{po}$ RW

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo'}\}+$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

define $\xrightarrow{sxo}$**:** X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ R

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$**:**

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all *competing* sub-operations.

Figure 15: Aggressive conditions for RCpc.

Appendix C: Proofs of Conditions in Appendix B

This appendix shows that the aggressive conditions for the different models presented in Appendix B are equivalent to the more conservative conditions that were used to define these models in the earlier literature. The latter part of the appendix provides the conservative conditions for PC, WO, RCsc, and RCpc expressed in our framework. These conditions are effectively a direct translation of the original definitions for the models (see Appendix B for further explanation).

C.1: Proofs for SC

This section shows that the aggressive conditions for SC (as in Figure 4) and Lamport's definition of SC [Lam79] are equivalent. Lamport's definition, as interpreted in previous work [AH90b, GLL$^+$90, GMG91], requires every execution on a sequentially consistent system to obey the following.

> There exists a total order, $\xrightarrow{to}$, on the memory operations of the execution such that (i) if A $\xrightarrow{po}$ B then A$\xrightarrow{to}$ B, and (ii) a read returns the value of the last write ordered before it by $\xrightarrow{to}$ that is to the same location as the read.

Theorem 1 states that a system that obeys the aggressive conditions for SC also obeys Lamport's definition. Theorem 2 states that a system that obeys Lamport's definition also obeys the aggressive conditions.

Theorem 1: A system that obeys the aggressive conditions for SC (Figure 4) also obeys Lamport's definition of SC.

Proof: Consider any execution, E_a, on a system that obeys the aggressive conditions for SC. The following shows that there must exist a total order on the operations of E_a that obeys Lamport's condition, thus proving theorem 1.

The proof consists of three steps. The first step shows in lemma 1.1 that E_a has an $\xrightarrow{xo}$ that obeys the aggressive conditions of SC with an augmented uniprocessor dependence condition. The second step shows in lemma 1.2 that E_a has an $\xrightarrow{xo}$ that obeys the conditions of lemma 1.1 with an augmented $\xrightarrow{spo}$ relation. The third step shows in lemma 1.3 that the $\xrightarrow{xo}$ demonstrated by lemma 1.2 implies that E_a obeys Lamport's definition.

Lemma 1.1: There is an $\xrightarrow{xo}$ of E_a that obeys the aggressive condition of SC with a modification to the uniprocessor dependence part of the definition of $\xrightarrow{sxo}$ as follows:

> *Define $\xrightarrow{sxo}$: X $\xrightarrow{sxo}$ Y if X and Y conflict and X,Y are the first and last operations in one of:*
> *uniprocessor dependence: RW $\xrightarrow{po}$ RW*
> *coherence, multiprocessor dependence chain, atomic read-modify-write : same as before*

Proof: Consider an $\xrightarrow{xo}$ of E_a that obeys the aggressive conditions of SC. Modify this $\xrightarrow{xo}$ as follows: First, for every R(i) such that W $\xrightarrow{po}$ R and R(i) $\xrightarrow{xo}$ W(i), move R(i) to immediately after the last such W(i) in the $\xrightarrow{xo}$ order. (The uniprocessor correctness condition ensures that there are only a finite number of such writes.) Second, move every W_{init}(i) to immediately before its corresponding W(i) in the $\xrightarrow{xo}$ order.

We first show that the modified order is an execution order for E_a; i.e., it obeys the value condition. Thus, we need to show that the value condition applied to the modified order requires a read to return the same value as it does when applied to the original $\xrightarrow{xo}$. First consider a read, R(i), that is not moved in the first step of the modification. The value condition applied to the original $\xrightarrow{xo}$ and applied to the modified order require that R(i) return the value of the conflicting write W(i) that is ordered last before it by the corresponding order. This W(i) is the same for both the orders. Thus, the value condition requires R(i) to return the same value with the modified order as with the original $\xrightarrow{xo}$. Now consider a read, R(i), that is moved in the first step of the modification. The value condition and the initiation, and uniprocessor dependence part of the $\xrightarrow{sxo}$ conditions of the aggressive conditions for SC applied to the original $\xrightarrow{xo}$ require that the read return the value of the last conflicting write, W(i), before it by program order. The modification ensures that W(i) is the last conflicting write before R(i) in the modified order, and W_{init}(i) is before R(i). Thus, the value condition applied to the modified order would require R to return the value of W as well. Thus, all reads are required to return the same value with the modified order as with the original order. Thus, the modified order obeys the value condition and is an execution order. We will refer to this order as the modified $\xrightarrow{xo}$.

The modified $\xrightarrow{xo}$ also clearly obeys the uniprocessor dependence part of the $\xrightarrow{sxo}$ condition using the modified $\xrightarrow{sxo}$ relation given by the statement of lemma 1.1. Similarly, it also obeys the coherence part and the read-modify-write part of the $\xrightarrow{sxo}$ condition. We show next that it obeys the multiprocessor dependence chain part of the $\xrightarrow{sxo}$ condition below, thus proving the lemma.

Assume for a contradiction that the modified $\xrightarrow{xo}$ does not obey the multiprocessor dependence chain part of the $\xrightarrow{sxo}$ condition; i.e., there is a multiprocessor dependence chain from X to Y with the modified $\xrightarrow{xo}$ but Y(i) is ordered before X(i) for some i by the modified $\xrightarrow{xo}$. There are three types of chains and each is discussed below. We use the fact that two conflicting sub-operations A(j) and B(j) can be ordered differently by the original and the modified $\xrightarrow{xo}$ only if A(j) is a read, B(j) is a write, A(j) $\xrightarrow{xo}$ B(j) in the original $\xrightarrow{xo}$, and there is a W(j) such that W(j) $\xrightarrow{po}$ A(j) and A(j) $\xrightarrow{xo}$ W(j) $\xrightarrow{xo}$ B(j) in the original $\xrightarrow{xo}$.

Case a: X=W $\xrightarrow{co'}$ R' $\xrightarrow{po}$ Y=R.

There are two sub-cases discussed below.

Sub-case a1: X(i) is ordered before Y(i) by the original $\xrightarrow{xo}$.

Since X is a write and Y is a read, X(i) must be ordered before Y(i) in the new $\xrightarrow{xo}$ also. This is a contradiction.

Sub-case a2: X(i) is ordered after Y(i) by the original $\xrightarrow{xo}$.

Thus, it must be that R'(i) was before W(i) in the original $\xrightarrow{xo}$; otherwise, there would have been a multiprocessor dependence chain from X to Y, and so X(i) should have been before Y(i) in the original $\xrightarrow{xo}$. Thus, R'(i) was moved below W(i) in the modification of the original $\xrightarrow{xo}$. But then Y(i) should also have been moved below W(i), a contradiction.

Case b: X $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ Y.

There are two sub-cases discussed below.

Sub-case b1: X(i) was after Y(i) in the original $\xrightarrow{xo}$.

The above chain from X to Y could not have existed in the original $\xrightarrow{xo}$ (otherwise the $\xrightarrow{sxo}$ condition would require X(i) to be before Y(i)). Therefore, one or more of the $\xrightarrow{sco}$ arcs in the above chain are of the type W $\xrightarrow{sco}$ R where R(i) was moved after W(i) in the modification of the original $\xrightarrow{xo}$.

Consider one of the W $\xrightarrow{sco}$ R arcs mentioned above. Let R return the value of W' in E_a. Then W(i) must be before W'(i) in the original $\xrightarrow{xo}$. So we can replace each of the W $\xrightarrow{sco}$ R arcs of the above type with a W sco W' arc to get a chain from X to Y that was also present in the original $\xrightarrow{xo}$. But then by $\xrightarrow{sxo}$ condition on the original $\xrightarrow{xo}$, X(i) must have been before Y(i) in the original $\xrightarrow{xo}$, a contradiction.

Sub-case b2: X(i) was before Y(i) in the original $\xrightarrow{xo}$.

X(i) must have been moved to after Y(i) during the modification. So X is a read and Y is a write. Let X return the value of the write W' in E_a. Since X(i) was moved after Y(i), W'(i) must be after Y(i) by the original (and hence the modified) $\xrightarrow{xo}$. Thus, there is a multiprocessor dependence chain from W' to Y in the modified $\xrightarrow{xo}$ that is of the type discussed in sub-case b1. As proved in sub-case b1, W'(i) must be before Y(i) in the modified $\xrightarrow{xo}$, a contradiction.

Case c: X=W1 $\xrightarrow{sco}$ R1 $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ Y=R2.

If the above chain exists in the original $\xrightarrow{xo}$ as well, then X(i) is before Y(i) in the original $\xrightarrow{xo}$ and hence in the new $\xrightarrow{xo}$, a contradiction. Therefore, there must be one or more W $\xrightarrow{sco}$ R arcs on the above chain where R(i) was moved after W(i) by the modification of $\xrightarrow{xo}$. We call such an arc a violating arc for the rest of this case. There are two sub-cases.

Sub-case c1: The first arc on the chain is not a violating arc.

As in Case b, each violating W $\xrightarrow{sco}$ R arc can be replaced by a W $\xrightarrow{sco}$ W' arc where W' is the write whose value is returned by R leading to a multiprocessor dependence chain from X to Y that must be present in the original $\xrightarrow{xo}$. Thus, X(i) is before Y(i) in the original $\xrightarrow{xo}$ and so it must be in the new $\xrightarrow{xo}$ too, a contradiction.

Sub-case c2: The first arc on the chain is a violating arc.

Let W' be the write whose value is returned by R1. Then there is a multiprocessor dependence chain of the type discussed in case b from W' to R2 with the new $\xrightarrow{xo}$. So by case b, it follows that W'(i) is before R2(i) in the new $\xrightarrow{xo}$. It must be that W1(i) is before W'(i) in the original (and so in the new) $\xrightarrow{xo}$. Thus, W1(i)=X(i) is before R2(i)=Y(i) in the new $\xrightarrow{xo}$, a contradiction.

The three cases complete the proof of lemma 1.1. □

Lemma 1.2: There is an $\xrightarrow{xo}$ of E_a that obeys the aggressive condition of SC with the modification of the uniprocessor dependence part of the $\xrightarrow{sxo}$ definition as stated in lemma 1.1 and with the following modification to the $\xrightarrow{spo}$ relation of the aggressive conditions:

$X \xrightarrow{spo} Y$ if $X \xrightarrow{po} Y$ (i.e., X and Y do not have to be from different locations)

Proof: Consider an $\xrightarrow{xo}$ of E_a that obeys the conditions of lemma 1.1, and consider the corresponding $\xrightarrow{co'}$ relation.

For a contradiction, assume the lemma is not true. Then there must be operations X,Y such that there is a multiprocessor dependence chain of the second or third type from X to Y with the modified definition of $\xrightarrow{spo}$ and Y(i) $\xrightarrow{xo}$ X(i). On this chain, there must be at least one $\xrightarrow{spo}$ arc that is due to the modification; i.e., the $\xrightarrow{spo}$ arc is between operations to the same location. This implies that there is a cycle in $\xrightarrow{po} \cup \xrightarrow{co'}$ and at least one of the $\xrightarrow{po}$ arcs on the cycle is between operations to the same location. Consider the shortest such cycle in terms of the number of $\xrightarrow{po}$ and $\xrightarrow{co'}$ arcs. Let one of the $\xrightarrow{po}$ arcs between accesses to the same location in this cycle be U $\xrightarrow{po}$ V. Then the cycle must be of one of two forms which are analyzed as separate cases below.

Case a: The cycle is of the form $A \xrightarrow{co'} U \xrightarrow{po} V \xrightarrow{co'} B$ ... A, where A != B, and U and V are to the same location.

There are three sub-cases.

Sub-case a1: A and B are from the same processor.

Then the path from A to B on the cycle above can be replaced by one $\xrightarrow{po}$ arc. This $\xrightarrow{po}$ arc is between operations to the same location and so we have a cycle in $\xrightarrow{po} \cup \xrightarrow{co'}$ such that at least one $\xrightarrow{po}$ arc is between operations to the same location, and this cycle is shorter than the chosen cycle, a contradiction.

Sub-case a2: A and B are from different processors, and there is a $\xrightarrow{po}$ arc between operations to the same location on the path from B to A on the cycle.

At least one of A, U, V, and B must be a write. Thus, either A $\xrightarrow{co'}$ B, or A $\xrightarrow{co'}$ U $\xrightarrow{co'}$ B, or A $\xrightarrow{co'}$ V $\xrightarrow{co'}$ B can replace the path from A to B on the cycle. Thus, again we have a cycle in $\xrightarrow{po} \cup \xrightarrow{co'}$ that has at least one $\xrightarrow{po}$ arc between operations to the same location and that is shorter than the chosen cycle, a contradiction.

Sub-case a3: A and B are from different processors, and there is no $\xrightarrow{po}$ arc between operations to the same location on the path from B to A on the cycle.

Either A and B conflict or A and B are both reads. Each case is handled separately in the paragraphs below.

Suppose A and B conflict. Then the path from B to A is either a multiprocessor dependence chain (by the modification of lemma 1.1) or the path consists of a multiprocessor dependence chain from B to a write W followed by W co A. In either case, B(i) $\xrightarrow{xo}$ A(i) for all i. However, the first three arcs on the cycle require A $\xrightarrow{co}$ B and so A(i) $\xrightarrow{xo}$ B(i) for all i, a contradiction.

Suppose A and B are both reads. Then V must be a write. Either the path from V to A is a multiprocessor dependence chain (by the definition of lemma 1.1) or the path consists of a multiprocessor dependence chain from V to a write W followed by W co A. In either case, V(i) $\xrightarrow{xo}$ A(i) for all i. However, the first two arcs on the cycle imply that A(i) $\xrightarrow{xo}$ V(i) for all i, a contradiction.

Case b: The cycle is of the form A $\xrightarrow{co'}$ *U* $\xrightarrow{po}$ *V* $\xrightarrow{co'}$ *B = A, where U and V are to the same location.* There are two sub-cases discussed below.

Sub-case b1: U and V conflict.

By the uniprocessor dependence part (as modified by lemma 1.1) of the $\xrightarrow{sxo}$ condition, it follows that U(i) $\xrightarrow{co}$ V(i) for all i. Since A and V conflict, it follows (using the coherence part of the $\xrightarrow{sxo}$ condition) that A(i) $\xrightarrow{xo}$ V(i) for all i. The cycle implies V(i) $\xrightarrow{xo}$ A(i) for all i, a contradiction.

Sub-case b2: U and V are both reads.

A must be a write. So the path from A to V is a multiprocessor dependence chain of the first type according to the original definition of such chains specified in Figure 4. Thus, A(i) $\xrightarrow{xo}$ V(i) for all i. However, the cycle requires V(i) $\xrightarrow{xo}$ A(i) for all i, a contradiction.

The above two cases complete lemma 1.2. □

Lemma 1.3: E_a obeys Lamport's definition of SC.

Proof: Consider the $\xrightarrow{xo}$ (and corresponding $\xrightarrow{co}$) guaranteed by lemma 1.2. E_a obeys Lamport's definition of SC if there is a total order on the memory operations of E_a that is consistent with $\xrightarrow{po} \cup \xrightarrow{co'}$. Therefore, for a contradiction assume that there is no such total order. Then there must be a cycle in $\xrightarrow{po} \cup \xrightarrow{co'}$. But this cycle implies a violation of the $\xrightarrow{sxo}$ condition (with $\xrightarrow{sxo}$ as modified by lemma 1.2), a contradiction. □

The proof of theorem 1 follows from lemma 1.3. ■

Theorem 2: A system that obeys Lamport's definition of SC also obeys the aggressive conditions for SC (Figure 4).

Proof: Consider an execution, E_c on a system that obeys Lamport's definition of SC. Consider the total order, $\xrightarrow{to}$, on the memory operations of E_c that is guaranteed by Lamport's definition. Derive a corresponding total order, $\longrightarrow$, on memory sub-operations as follows: X(i) $\longrightarrow$ Y(j) for all i,j iff X $\xrightarrow{to}$ Y, and W_{init}(i) is immediately before W(i) by $\longrightarrow$ for any write sub-operation W(i) issued by any processor Pi. Then it follows that $\longrightarrow$ is an $\xrightarrow{xo}$ and this $\xrightarrow{xo}$ obeys the aggressive conditions of SC. ■

C.2: Proofs for TSO

This section shows that the aggressive (Figure 11) and original conditions for TSO [SFC91] are equivalent. We will refer to the original conditions as the conservative conditions below. Theorem 3 states that a system that obeys the aggressive conditions for TSO also obeys the conservative conditions for TSO. Theorem 4 states that a system that obeys the conservative conditions for TSO also obeys the aggressive conditions.

Theorem 3: A system that obeys the aggressive conditions for TSO (Figure 11) also obeys the conservative conditions for TSO [SFC91].

Proof:

Consider any execution, E_a, that has an $\xrightarrow{xo}$ that obeys the aggressive conditions for TSO. The following shows that there must exist a total order on the operations of E_a that obeys all the axioms of the conservative conditions for TSO, thus proving theorem 3. Below, the relation $\xrightarrow{xo}$ implicitly refers to the $\xrightarrow{xo}$ of E_a that obeys the aggressive conditions for TSO. The relations $\xrightarrow{co}$, $\xrightarrow{co'}$, and $\xrightarrow{sco}$ refer to the corresponding relations for the above $\xrightarrow{xo}$.

The proof consists of four steps. The first step constructs a relation $\xrightarrow{rel}$ on the operations of E_a. The second step shows in Lemma 3.1 that the relation $\xrightarrow{rel}$ is a partial order. The third step shows in Lemma 3.2 that any total order consistent with $\xrightarrow{rel}$ obeys all the axioms of the conservative conditions of TSO except possibly the atomicity axiom. The final step shows in Lemma 3.3 that there is at least one total order consistent with $\xrightarrow{rel}$ that also obeys the atomicity axiom of the conservative conditions of TSO. Thus, it follows that there is a total order for E_a that obeys all the axioms of the conservative conditions of TSO, completing the proof.

Construction of relation $\xrightarrow{rel}$

Define $\xrightarrow{rel}$: X $\xrightarrow{rel}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{spo}$ Y
W $\xrightarrow{co'}$ Y
R $\xrightarrow{co'}$ W and if R returns the value of its processor's write, W', in E_a, then W' $\xrightarrow{co'}$ W
X $\xrightarrow{rel}$ Z $\xrightarrow{rel}$ Y

Lemma 3.1: Relation $\xrightarrow{rel}$ *is a partial order.*

Proof: For a contradiction, assume $\xrightarrow{rel}$ is not a partial order. Then there must be a cycle C involving the $\xrightarrow{spo}$ and $\xrightarrow{co'}$ arcs that make up $\xrightarrow{rel}$. We first show in Claim 3.1 below that cycle C implies a cycle C' in $\xrightarrow{spo} \cup \xrightarrow{sco}$ with alternating $\xrightarrow{spo}$ and $\xrightarrow{sco}$ arcs. Furthermore, if R $\xrightarrow{sco}$ W is one of the $\xrightarrow{sco}$ arcs on C', then it cannot be that W $\xrightarrow{spo}$ R. We then show in Claim 3.2 that this cycle C' contradicts the fact that $\xrightarrow{xo}$ obeys the aggressive conditions for TSO. This contradiction completes the proof of Lemma 3.1.

Claim 3.1: A cycle C in $\xrightarrow{rel}$ implies a cycle C' in $\xrightarrow{spo} \cup \xrightarrow{sco}$ with alternating $\xrightarrow{spo}$ and $\xrightarrow{sco}$ arcs. Furthermore, if R $\xrightarrow{sco}$ W is one of the $\xrightarrow{sco}$ arcs on cycle C', then it cannot be that W $\xrightarrow{spo}$ R.

Proof: Consider a cycle C in $\xrightarrow{rel}$ involving the fewest number of $\xrightarrow{spo}$ and $\xrightarrow{co'}$ arcs.

First we show that there cannot be two consecutive $\xrightarrow{spo}$ arcs on cycle C. For a contradiction, assume that there are two such arcs: X $\xrightarrow{spo}$ Y $\xrightarrow{spo}$ Z. Then it cannot be that X $\xrightarrow{spo}$ Z; otherwise, the two $\xrightarrow{spo}$ arcs can be replaced by one $\xrightarrow{spo}$ arc leading to a shorter cycle than C in $\xrightarrow{rel}$, a contradiction. Therefore, it must be that X is a write and Z is a read, X is not from a RMW and there is no RMW such that X $\xrightarrow{po}$ RMW $\xrightarrow{po}$ Z. But then the only way for X $\xrightarrow{spo}$ Y to be possible is for Y to be a write. But then it cannot be that Y $\xrightarrow{spo}$ Z, a contradiction. Thus, each $\xrightarrow{spo}$ arc on cycle C must be preceded and followed by a $\xrightarrow{co'}$ arc.

It is straightforward to show that a chain of consecutive $\xrightarrow{co'}$ arcs can always be replaced by a $\xrightarrow{co}$ arc or a chain of the form R $\xrightarrow{co}$ W $\xrightarrow{co}$ R. Thus, a chain of consecutive $\xrightarrow{co'}$ arcs on the cycle C can be replaced by a $\xrightarrow{sco}$ arc.

From the above two paragraphs it follows that the cycle C also implies a cycle C' of alternating $\xrightarrow{spo}$ and $\xrightarrow{sco}$ arcs, proving the first part of claim 3.1. We now prove the second part of the claim. For a contradiction, assume one of the $\xrightarrow{sco}$ arcs on cycle C' (obtained from cycle C as shown in the above two paragraphs) is R $\xrightarrow{sco}$ W where W $\xrightarrow{spo}$ R. Then it must be that this $\xrightarrow{sco}$ arc was formed by replacing a chain of consecutive $\xrightarrow{co'}$ arcs in cycle C. This implies that in cycle C there must be a chain of the type R $\xrightarrow{co'}$ W' $\xrightarrow{co'}$ ... $\xrightarrow{co'}$ W. Furthermore, since R $\xrightarrow{co}$ W, the value condition and the uniprocessor dependence part of the sxo conditions of the aggressive conditions of TSO ensure that R will return the value of its own processor's write in E_a. However, then the definition of $\xrightarrow{rel}$ implies that R $\xrightarrow{co'}$ W' cannot be on the cycle C, a contradiction. Thus, if R $\xrightarrow{sco}$ W is on cycle C', then it cannot be that W $\xrightarrow{spo}$ R, proving the claim. □

Claim 3.2: The existence of cycle C' described by Claim 3.1 contradicts the condition that $\xrightarrow{xo}$ *obeys the aggressive conditions for TSO.*

Proof: Let A1 $\xrightarrow{spo}$ A2 $\xrightarrow{sco}$ A3 $\xrightarrow{spo}$ A4 ... An $\xrightarrow{sco}$ A1 be the cycle C' described by claim 3.1. Note that n must be even and greater than 0. There are two cases depending on whether A1 and An conflict. We show below that each case obeys claim 3.2.

Case a: A1 and An conflict.

We first consider the sub-case where n = 2 and then the sub-case where n $\geq$ 4, and show that both sub-cases obey the claim. If n = 2, then claim 3.1 implies that it cannot be that A1 is a write and A2 is a read. Thus, the above cycle implies that $\xrightarrow{xo}$ contradicts the uniprocessor dependence part of the $\xrightarrow{sxo}$ condition, proving claim 3.2. If n $\geq$ 4, then the above cycle implies that $\xrightarrow{xo}$ contradicts the multiprocessor dependence chain part of the $\xrightarrow{sxo}$ condition for the chain from A1 to An, again proving claim 3.2.

Case b: A1 and An do not conflict.

There must be a W such that A1=R $\xrightarrow{spo}$ A2 $\xrightarrow{sco}$ A3 $\xrightarrow{spo}$ A4 ... An=R $\xrightarrow{sco}$ W $\xrightarrow{sco}$ A1=R. Alternately, W $\xrightarrow{sco}$ A1=R $\xrightarrow{spo}$ A2 $\xrightarrow{sco}$ A3 $\xrightarrow{spo}$ A4 ... An=R $\xrightarrow{sco}$ W. This cycle implies that $\xrightarrow{xo}$ contradicts the multiprocessor dependence chain part of the $\xrightarrow{sxo}$ condition, proving claim 3.2.

The two cases above complete the proof of Claim 3.2. □

Claims 3.1 and 3.2 complete the proof of lemma 3.1. □

Lemma 3.2: Any total order on the operations of E_a that is consistent with $\xrightarrow{rel}$ obeys all the axioms of the conservative condition for TSO except possibly the axiom of atomicity.

Proof: Consider a total order $\xrightarrow{to}$ on the operations of E_a that is consistent with $\xrightarrow{rel}$. This order obeys the Order, LoadOp, and StoreStore axioms of the conservative conditions for TSO by construction. The order also obeys the Termination axiom of the conservative conditions of TSO because of the following. The termination condition for TSO requires that a write appear after a finite number of reads to the same location. Below, we show that the order also obeys the Value axiom of the conservative conditions for TSO; i.e., if R returns the value of W in E_a, then the value axiom of the conservative conditions of TSO when applied to $\xrightarrow{to}$ would require that R return the value of W. There are two cases depending on whether R returns the value of its own processor's write or another processor's write and we show the above must be true in each case. Each case uses the fact that for two conflicting writes W1 and W2, if W1 $\xrightarrow{co}$ W2, then W1 $\xrightarrow{rel}$ W2 and so W1 $\xrightarrow{to}$ W2. We also use the fact that the termination condition ensures that conflicting operations are related by $\xrightarrow{co}$.

Case a: R returns the value of another processor's write, W, in E_a.

$\xrightarrow{rel}$ (and therefore $\xrightarrow{to}$) orders R with respect to all writes of other processors that conflict with R, and this order is the same as by $\xrightarrow{co}$. Since all conflicting writes are similarly ordered by $\xrightarrow{to}$ and $\xrightarrow{co}$, it follows that the last conflicting write of another processor before R by $\xrightarrow{to}$ is W. We know by the value condition and the initiation condition of the aggressive conditions of TSO that all conflicting writes ordered before R by program order must be ordered before W by $\xrightarrow{co}$ (otherwise, R would have returned the value of its own processor's write). Therefore, again, since all conflicting writes are ordered similarly by $\xrightarrow{to}$ and $\xrightarrow{co}$, all conflicting writes before R by program order must be ordered before W (and so also before R) by $\xrightarrow{to}$. Thus, the value axiom of the conservative conditions of TSO applied to $\xrightarrow{to}$ requires that R return the value of W.

Case b: R returns the value of its own processor's write W in E_a.

The only case when the value axiom of the conservative conditions for TSO when applied to $\xrightarrow{to}$ would require that R return the value of another write, W', where W' $\neq$ W is if W $\xrightarrow{to}$ W' $\xrightarrow{to}$ R and W' is from another processor. But in this case, W $\xrightarrow{co}$ W' and so $\xrightarrow{rel}$ ensures that R $\xrightarrow{to}$ W', a contradiction. Therefore, the value axiom of the conservative conditions of TSO also requires R to return the value of its own processor's write and this write must be W.

Thus, $\xrightarrow{to}$ obeys the value axiom of the conservative conditions as well, proving lemma 3.2. □

Lemma 3.3: There exists a total order on the operations of E_a that is consistent with $\xrightarrow{rel}$ and that obeys the atomicity axiom of the conservative condition for TSO.

Proof:

Consider a RMW operation in E_a. Below we show in Claim 3.3 that there is no other operation ordered after the AR and before the AW of the RMW by $\xrightarrow{rel}$. This implies there exists a total order of the operations of E_a

that is consistent with $\xrightarrow{rel}$ and such that there is no operation between the AR and the AW of any RMW ordered by this total order. This total order obeys the atomicity axiom proving lemma 3.3.

Claim 3.3: For a RMW in E_a, there is no other operation ordered after the AR and before the AW of the RMW by $\xrightarrow{rel}$.

Proof: Suppose there is an operation, say X, between AR and AW ordered by $\xrightarrow{rel}$. Then consider some path from AR to X to AW consisting of the $\xrightarrow{spo}$ and $\xrightarrow{co'}$ arcs that make up $\xrightarrow{rel}$. There are two cases and we show below that each leads to a contradiction.

Case a. The first arc on the path from AR to X to AW is an $\xrightarrow{spo}$ arc, AR $\xrightarrow{spo}$ Y, where Y may or may not be the same as X.

We have either AR $\xrightarrow{spo}$ Y $\xrightarrow{rel}$ X $\xrightarrow{rel}$ AW or AR $\xrightarrow{spo}$ Y=X $\xrightarrow{rel}$ AW. It must be that AW $\xrightarrow{po}$ Y and so AW $\xrightarrow{spo}$ Y and so AW $\xrightarrow{rel}$ Y. Thus, we have a cycle in $\xrightarrow{rel}$ because either X $\xrightarrow{rel}$ AW $\xrightarrow{rel}$ Y $\xrightarrow{rel}$ X or X $\xrightarrow{rel}$ AW $\xrightarrow{rel}$ Y=X. This contradicts lemma 3.1 and is not possible.

Case b. The first arc on the path from AR to X to AW is a $\xrightarrow{co'}$ arc, AR $\xrightarrow{co'}$ W', where W' may or may not be the same as X.

We have either AR $\xrightarrow{co'}$ W' $\xrightarrow{rel}$ X $\xrightarrow{rel}$ AW or AR $\xrightarrow{co'}$ W'=X $\xrightarrow{rel}$ AW. Either W' $\xrightarrow{co'}$ AW or AW $\xrightarrow{co'}$ W'. We show that both cases lead to contradictions. If W' $\xrightarrow{co'}$ AW, then we have AR $\xrightarrow{co}$ W' $\xrightarrow{co}$ AW, but this implies that $\xrightarrow{xo}$ violates the atomicity condition of the aggressive conditions for TSO, a contradiction. If AW $\xrightarrow{co'}$ W', then we have a cycle in rel because either W' $\xrightarrow{rel}$ X $\xrightarrow{rel}$ AW $\xrightarrow{rel}$ W' or W'=X $\xrightarrow{rel}$ AW $\xrightarrow{rel}$ W', a contradiction to lemma 3.1.

The two cases above prove claim 3.3. □

This completes the proof of lemma 3.3. □

Theorem 3 follows from lemmas 3.2 and 3.3. ■

Theorem 4: A system that obeys the conservative conditions for TSO [SFC91] also obeys the aggressive conditions for TSO (Figure 11).

Proof: Consider any execution, E_c, that obeys the conservative conditions for TSO. The proof shows that there must exist a $\xrightarrow{xo}$ on the sub-operations of E_c that obeys the aggressive conditions for TSO, thereby proving the theorem.

Below, the relation $\xrightarrow{mo}$ refers to the partial order which is the memory order relation for E_c that obeys the different axioms of the conservative conditions for TSO.[29] Furthermore, $\xrightarrow{mo}$ refers to the minimal such partial order; i.e., its transitive reduction must consist of only arcs necessary to obey the axioms of the conservative conditions for TSO. Therefore, the transitive reduction consists of only arcs of the following types: (i) arcs between two writes, (ii) arcs from a read to any operation where the first read is before the second operation by program order, (iii) arcs from a write to a read where the write is before the read by program order, (iv) arcs from a write to a read where the read returns the value of the write, and (v) arcs between a read of a RMW and a write.

Consider relation $\xrightarrow{rel}$ on the operations of E_c as follows: R $\xrightarrow{rel}$ W if R returns the value of write W' in E_c and W' $\xrightarrow{mo}$ W.

The proof now proceeds in four steps. The first step shows in Lemma 4.1 that $\xrightarrow{rel} \cup \xrightarrow{mo}$ is acyclic. The second step shows in Lemma 4.2 that any total order consistent with $\xrightarrow{rel} \cup \xrightarrow{mo}$ (extended to sub-operations) on the sub-operations of E_c is an $\xrightarrow{xo}$. The third step shows in Lemma 4.3 that any $\xrightarrow{xo}$ of the type discussed in Lemma 4.2 obeys all the aggressive conditions for TSO except possibly the read-modify-write part of the $\xrightarrow{sxo}$ condition. The final step shows in Lemma 4.4 that there is at least one $\xrightarrow{xo}$ of the type discussed in Lemma 4.2 that also obeys the read-modify-write part of the $\xrightarrow{sxo}$ condition. Thus, it follows that there is an $\xrightarrow{xo}$ for E_a that obeys the aggressive conditions, completing the proof.

[29]We assume this partial order obeys the property in footnote 6: there are only a finite number of operations ordered before any given operation.

Lemma 4.1: $\xrightarrow{rel} \cup \xrightarrow{mo}$ *is acyclic.*

Proof: Suppose there is a cycle in $\xrightarrow{rel} \cup \xrightarrow{mo}$. Consider the shortest such cycle. It must be of the type: R1 $\xrightarrow{rel}$ W1 $\xrightarrow{mo}$ R2 $\xrightarrow{rel}$ W2 ... Rn $\xrightarrow{rel}$ Wn $\xrightarrow{mo}$ R1, where n $\geq$ 1. There are two cases depending on the value of n and both lead to contradictions as discussed below.

Case a: n is $\geq$ 2.
Either W1 $\xrightarrow{mo}$ W2 or W2 $\xrightarrow{mo}$ W1. Suppose W1 $\xrightarrow{mo}$ W2. Then if n is greater than 2, W1 $\xrightarrow{mo}$ R2 $\xrightarrow{rel}$ W2 $\xrightarrow{mo}$ R3 can be replaced by W1 $\xrightarrow{mo}$ R3, resulting in a shorter cycle, a contradiction. If n = 2, then W1 $\xrightarrow{mo}$ R2 $\xrightarrow{rel}$ W2 $\xrightarrow{mo}$ R1 can be replaced by W1 $\xrightarrow{mo}$ R1, also resulting in a shorter cycle, a contradiction. Thus, the case of W1 $\xrightarrow{mo}$ W2 is not possible. Now suppose W2 $\xrightarrow{mo}$ W1. Then we have W2 $\xrightarrow{mo}$ R2 and so we have a shorter cycle, R2 $\xrightarrow{rel}$ W2 $\xrightarrow{mo}$ R2, a contradiction, completing this case.

Case b: n = 1.
We have R1 $\xrightarrow{rel}$ W1 $\xrightarrow{mo}$ R1. If R1 returned the value of W' in E_c then W' $\xrightarrow{mo}$ W1 $\xrightarrow{mo}$ R1. But this contradicts the value axiom of the conservative conditions.

The above two cases prove lemma 4.1. □

Lemma 4.2: Consider a total order $\xrightarrow{to'}$ on the operations of E_c that is consistent with $\xrightarrow{mo} \cup \xrightarrow{rel}$. Define $\xrightarrow{to}$ on the sub-operations of E_c as follows. X(i) $\xrightarrow{to}$ Y(j) iff X $\xrightarrow{to'}$ Y. Further, for a write W(i) issued by Pi, order its corresponding $W_{init}(i)$ sub-operation as follows. If there exists an R such that W $\xrightarrow{po}$ R and R $\xrightarrow{to'}$ W, then let $W_{init}(i)$ be immediately before the first such R(i) in $\xrightarrow{to}$. If there does not exist the above type of R, then let $W_{init}(i)$ be immediately before W(i) in $\xrightarrow{to}$. Then the order $\xrightarrow{to}$ is an $\xrightarrow{xo}$.

Proof: To show that $\xrightarrow{to}$ is an $\xrightarrow{xo}$, we need to show that it obeys the value condition. Consider a read R(i) that returns the value of W(i) in E_c. For a contradiction, assume that the value condition of the aggressive conditions for TSO applied to $\xrightarrow{to}$ requires that R return the value of another write, W'. There are three cases possible discussed below, and each leads to a contradiction as shown. (Each of the cases uses the fact that conflicting writes are ordered by $\xrightarrow{to}$ in the same way as by $\xrightarrow{mo}$.)

Case a: W $\xrightarrow{mo}$ W'.
Then R $\xrightarrow{rel}$ W' and so R(i) $\xrightarrow{to}$ W'(i). Further, it cannot be that W' $\xrightarrow{po}$ R; otherwise, R would have returned the value of W' in E_c. So the value condition of the aggressive condition for TSO applied to $\xrightarrow{to}$ cannot require R to return the value of W', a contradiction.

Case b: W' $\xrightarrow{mo}$ W and W $\xrightarrow{mo}$ R.
Then W'(i) $\xrightarrow{to}$ W(i) $\xrightarrow{to}$ R(i) and so the value condition of the aggressive condition for TSO applied to $\xrightarrow{to}$ cannot require that R return the value of W', a contradiction.

Case c: W' $\xrightarrow{mo}$ W and $\neg$ (W $\xrightarrow{mo}$ R).
Then W is from the same processor as R. In either case of W(i) $\xrightarrow{to}$ R(i) or R(i) $\xrightarrow{to}$ W(i), it follows that the value condition of the aggressive condition for TSO applied to $\xrightarrow{to}$ would require that R not return the value of W'.

Thus, the value condition of the aggressive condition for TSO applied to $\xrightarrow{to}$ requires that R return the value of W, and so $\xrightarrow{to}$ obeys the value condition and is an $\xrightarrow{xo}$, proving Lemma 4.2. □

Lemma 4.3: Any total order $\xrightarrow{to}$ described in Lemma 4.2 obeys all the aggressive conditions for TSO except possibly the read-modify-write part of the $\xrightarrow{sxo}$ condition.

Proof: $\xrightarrow{to}$ clearly obeys the initiation condition (by construction). It also obeys the termination condition and the uniprocessor dependence and coherence parts of the $\xrightarrow{sxo}$ condition of the aggressive condition for TSO since similar (or more conservative) conditions are obeyed by $\xrightarrow{mo}$. It also obeys the termination condition by construction. We next show that it obeys the three parts of the multiprocessor dependence chain part of the $\xrightarrow{sxo}$ condition.

We first discuss the first part of the multiprocessor dependence chain. For this part, we need to consider a chain of the type W $\xrightarrow{to'}$ R1 $\xrightarrow{spo}$ R2, where all three operations on the trace conflict. R1 $\xrightarrow{spo}$ R2 implies R1 $\xrightarrow{mo}$ R2 and so R1 $\xrightarrow{to'}$ R2. So W $\xrightarrow{to'}$ R2 and so W(i) $\xrightarrow{to}$ R(i) for all i. Thus, $\xrightarrow{to}$ obeys the first part of the dependence chain condition.

We next discuss the second part of the multiprocessor dependence chain condition. For this part, we need to consider chains of the following type: A1 $\xrightarrow{spo}$ A2 $\xrightarrow{sco}$ A3 $\xrightarrow{spo}$ A4 ... $\xrightarrow{spo}$ An, where A1 and An conflict. There are two cases and for each case we show below that A1(p) $\xrightarrow{to}$ An(q) for all p,q, thereby obeying the required condition.

Case a: All cases except when An-1 is a W from RMW and An is a read.

We show by induction that for the chain given above, for every k, either

Ak=A1, or

A1(p) $\xrightarrow{to}$ Ak(q) for all p,q, or

Ak is a read, Ak-1 is a write from a RMW, Ak-1 $\xrightarrow{spo}$ Ak, and if Ak+1 exists, then A1(p) $\xrightarrow{to}$ Ak+1(q) for all p,q.

Thus, it will follow that A1(p) $\xrightarrow{to}$ An(q) for all p,q as required.

Base case: Trivially true.

Induction: If Ak-1 $\xrightarrow{sco}$ Ak, then we are done by induction hypothesis. So suppose A1 ... $\xrightarrow{sco}$ Ak-1 $\xrightarrow{spo}$ Ak. By induction hypothesis, A1(p) $\xrightarrow{to}$ Ak-1(q) for all p,q. Ak-1(p) $\xrightarrow{to}$ Ak(q) for all cases except when Ak-1 is a W from RMW and Ak is a read. Thus, in all cases except the above, the proposition is true. So assume Ak-1 = W from RMW, Ak = R, Ak-1 $\xrightarrow{spo}$ Ak. If there is no Ak+1, then we are done. If there is an Ak+1, then either it is a write or a read. We show next that for both cases, the proposition is true. Suppose Ak+1 is a write. By the atomicity axiom, $\xrightarrow{mo}$ (and therefore $\xrightarrow{to'}$) either orders Ak+1 before the R or after the W of the RMW of Ak-1. Further, $\xrightarrow{mo}$ (and therefore $\xrightarrow{to'}$) orders Ak after the R of the RMW of Ak-1. Since Ak $\xrightarrow{sco}$ Ak+1, it follows that Ak-1 $\xrightarrow{to'}$ Ak+1. Therefore, Ak-1(p) $\xrightarrow{to}$ Ak(q) for all p,q and it follows that A1(p) $\xrightarrow{to}$ Ak+1(q) for all p,q. Thus, the proposition is true. Suppose Ak+1 is a read. Then there is a write W' such that Ak $\xrightarrow{co}$ W' $\xrightarrow{co}$ Ak+1. The above analysis for Ak+1=write applies to W' and so A1(p) $\xrightarrow{to}$ W'(q) for all p,q. Thus, A1(p) $\xrightarrow{to}$ Ak+1(q) for all p,q as well.

Case b: An-1 is a W from RMW and An is a read.

Then A1 must be a write and by Step 1, it must be ordered by $\xrightarrow{to}$ before the W of the RMW of An-1. By atomicity axiom on $\xrightarrow{mo}$, A1 must be ordered before the read of the RMW of An-1 by $\xrightarrow{mo}$ (and so by $\xrightarrow{to'}$). The R of the RMW of An-1 is ordered before An by $\xrightarrow{to'}$. Thus, A1(p) $\xrightarrow{to}$ An(q) for all p,q and so $\xrightarrow{to}$ obeys the second part of the dependence chain.

Finally, we consider the third part of the dependence chain. For this we need to consider the following types of chains: W sco A1=R $\xrightarrow{spo}$ A2 sco A3 $\xrightarrow{spo}$ A4 ... $\xrightarrow{spo}$ An=R The reasoning for the second part of the dependence chain can be directly applied here as well to show that this condition is also obeyed.

This completes the proof of lemma 4.3. □

Lemma 4.4: There is at least one $\xrightarrow{to}$ described by lemma 4.3 that obeys the read-modify-write part of the $\xrightarrow{sxo}$ condition of the aggressive conditions for TSO.

Proof: Consider a RMW operation in E_c. Below we show in claim 4.1 that there is no other operation ordered after the AR and before the AW of the RMW by $\xrightarrow{mo} \cup \xrightarrow{rel}$. This implies there exists a total order $\xrightarrow{to'}$ of the operations of E_c that is consistent with $\xrightarrow{mo} \cup \xrightarrow{rel}$ and such that there is no operation between the AR and the AW of any RMW ordered by this total order. Therefore, a total order $\xrightarrow{to}$ generated from $\xrightarrow{to'}$ on the sub-operations of E_c (as discussed in Lemma 4.2) obeys the read-modify-write part of the $\xrightarrow{sxo}$ condition, proving lemma 4.4.

Claim 4.1: For a RMW in E_c, there is no other operation ordered after the AR and before the AW of the RMW by $\xrightarrow{mo} \cup \xrightarrow{rel}$.

Proof: Suppose there is an operation, say X, between AR and AW ordered by $\xrightarrow{mo} \cup \xrightarrow{rel}$. Then consider some path from AR to X to AW consisting of the base arcs of $\xrightarrow{mo} \cup \xrightarrow{rel}$. (Base arcs are those that form the transitive reduction of $\xrightarrow{mo}$ and $\xrightarrow{rel}$ as discussed in the beginning of the proof of theorem 4.) There are two cases and both lead to contradictions.

Case a. X is a write.

Then we know that X must be ordered with respect to both AR and AW by $\xrightarrow{mo}$. But then it must be either after AW or before AR by $\xrightarrow{mo}$. Either case implies a cycle in $\xrightarrow{mo} \cup \xrightarrow{rel}$, a contradiction to lemma 4.1.

Case b. X is a read.

Case (a) has shown that there cannot be a write ordered between AR and AW by $\xrightarrow{rel} \cup \xrightarrow{mo}$. Thus, it follows that the path from AR to X and from X to AW cannot contain a write. Thus, the path must be of the type AR $\xrightarrow{mo}$ X=R $\xrightarrow{mo}$ AW, where AR $\xrightarrow{po}$ AW $\xrightarrow{po}$ X=R. But X=R $\xrightarrow{mo}$ AW is not a base arc for $\xrightarrow{mo}$, and so cannot be present, a contradiction.

The two cases prove claim 4.1. □

This completes the proof of lemma 4.4. □

Theorem 4 follows from lemmas 4.2, 4.3, and 4.4. ■

C.3: Proofs for PSO

This section shows that the aggressive (Figure 12) and original conditions for PSO [SFC91] are equivalent. We will refer to the original conditions as the conservative conditions below. Theorem 5 states that a system that obeys the aggressive conditions for PSO also obeys the conservative conditions for PSO. Theorem 6 states that a system that obeys the conservative conditions for PSO also obeys the aggressive conditions.

Theorem 5: A system that obeys the aggressive conditions for PSO (Figure 12) also obeys the conservative conditions for PSO [SFC91].

Proof: The proof of theorem 5 is very similar to that of theorem 3; i.e., it consists of constructing a relation $\xrightarrow{rel}$, and proving the statements of lemmas 3.1-3.3 with TSO and the old $\xrightarrow{rel}$ relation replaced by PSO and the new $\xrightarrow{rel}$ relation. The relation $\xrightarrow{rel}$ for PSO is similar to the relation $\xrightarrow{rel}$ for TSO except for an additional component of $\xrightarrow{po'}$ arcs defined below.

Define $\xrightarrow{po'}$: X $\xrightarrow{po'}$ Y if X $\xrightarrow{po}$ Y and X and Y are writes to the same location.

Define $\xrightarrow{rel}$: X $\xrightarrow{rel}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{spo}$ Y
W $\xrightarrow{co'}$ Y
R $\xrightarrow{co'}$ W and if R returns the value of its processor's write, W', in E_a, then W' $\xrightarrow{co'}$ W
X $\xrightarrow{po'}$ Y
X $\xrightarrow{rel}$ Z $\xrightarrow{rel}$ Y

The proofs for the lemmas corresponding to lemmas 3.1-3.3 for PSO are very similar to that for TSO. The only significant difference is in the proof of the first part of claim 3.1 in lemma 3.1 as follows. The first part of claim 3.1 requires proving that a cycle C in $\xrightarrow{rel}$ implies a cycle C' in $\xrightarrow{spo} \cup \xrightarrow{sco}$ consisting of alternating $\xrightarrow{spo}$ and $\xrightarrow{sco}$ arcs. As for TSO, we consider a cycle C in $\xrightarrow{rel}$ with the fewest number of $\xrightarrow{spo}$, $\xrightarrow{po'}$, and $\xrightarrow{co'}$ arcs. The next step is to show that there cannot be two consecutive $\xrightarrow{spo}$ arcs on cycle C. The reasoning for this is a straightforward extension of that for TSO that takes into account the differences in the $\xrightarrow{spo}$ arcs of TSO and PSO.

The final step is the straightforward observation that any chain of consecutive $\xrightarrow{co'}$ and $\xrightarrow{po'}$ arcs on cycle C can be replaced by a $\xrightarrow{co}$ arc or a chain of the form of R $\xrightarrow{co}$ W $\xrightarrow{co}$ R. (In the TSO case, we did not have to consider $\xrightarrow{po'}$ arcs.) Thus, any chain of consecutive $\xrightarrow{co'}$ and $\xrightarrow{po'}$ arcs on cycle C can be replaced by a $\xrightarrow{sco}$ arc. Thus, it follows that the cycle C implies a cycle of alternating $\xrightarrow{spo}$ and $\xrightarrow{sco}$ arc, proving the first part of claim 3.1 of lemma 3.1 for PSO. As mentioned above, the rest of the proof for PSO is almost identical to that for TSO. ■

Theorem 6: A system that obeys the conservative conditions for PSO [SFC91] also obeys the aggressive conditions for PSO 12.

Proof: This is virtually identical to the proof of theorem 4. ■

C.4: Proofs for PC, RCsc, and RCpc

The conservative and aggressive conditions for PC, RCsc, and RCpc follow a similar form and so we discuss these models together. This section shows that the aggressive conditions (Figures 10, 14, 15) and conservative conditions for each of the above models (Figures 16, 18, 19) are equivalent.

Theorem 7 gives a common proof for all the above models that shows that a system that obeys the aggressive conditions for any of the above models also obeys the conservative conditions of that model. Theorem 8 gives a common proof for all the above models that shows that a system that obeys the conservative conditions for any of the above models also obeys the aggressive conditions of that model.

Theorem 7: A system that obeys the aggressive conditions for PC, RCsc, and RCpc also obeys the conservative conditions for PC, RCsc, and RCpc respectively.

Proof: Let M represent any one of the models of PC, RCsc, and RCpc. Consider an execution, E_a, on a system that obeys the aggressive conditions for M. Consider an $\xrightarrow{xo}$ of E_a that obeys the aggressive conditions. The $\xrightarrow{xo}$ clearly obeys the initiation and termination parts of the conservative conditions. It obeys the $\xrightarrow{sxo}$ part of the conservative conditions because of the uniprocessor dependence aspect of the $\xrightarrow{sxo}$ part of the aggressive conditions. It obeys the $\xrightarrow{sxo''}$ part of the conservative conditions because of the coherence and read-modify-write aspects of the $\xrightarrow{sxo}$ part of the aggressive conditions. Thus, we only need to show that it also obeys the $\xrightarrow{sxo'}$ part of the conservative conditions.

We use the following two notions in our proof below. First, we overload the symbols $\xrightarrow{co}$ and $\xrightarrow{sxo'}$ to also imply relations between memory sub-operations as follows. A(i) $\xrightarrow{co}$ B(i) iff A $\xrightarrow{co}$ B, and A(i) $\xrightarrow{sxo'}$ B(j) iff A $\xrightarrow{sxo'}$ B. Second, we note that for an $\xrightarrow{xo}$ to obey the aggressive conditions, it is sufficient if the $\xrightarrow{co}$ (on sub-operations) corresponding to the $\xrightarrow{xo}$ is the same as the $\xrightarrow{co}$ of some other $\xrightarrow{xo}$ that is known to obey the aggressive conditions.

For a contradiction, assume that no $\xrightarrow{xo}$ of E_a that obeys the aggressive conditions of model M also obeys the $\xrightarrow{sxo'}$ part of the conservative conditions of model M. Now consider some $\xrightarrow{xo}$ of E_a that obeys the aggressive conditions of model M. Then it follows that for this $\xrightarrow{xo}$, there must be a cycle in $\xrightarrow{co} \cup \xrightarrow{sxo'}$ where $\xrightarrow{co}$ and $\xrightarrow{sxo'}$ are on sub-operations. (Otherwise, we can define an $\xrightarrow{xo}$ that has the same $\xrightarrow{co}$ as the $\xrightarrow{xo}$ considered above and such that if A $\xrightarrow{sxo'}$ B, then A(i) $\xrightarrow{xo}$ B(j) for all i,j. This $\xrightarrow{xo}$ obeys the conservative conditions of model M, a contradiction.)

Consider the shortest cycle in $\xrightarrow{co}$ union $\xrightarrow{sxo'}$ (in terms of the number of $\xrightarrow{co}$ and $\xrightarrow{sxo'}$ arcs). Let this cycle be A(i) ... B(i) $\xrightarrow{co}$ A(i). This cycle obeys the following properties.

(P7.1) The cycle has at most two consecutive $\xrightarrow{co}$ arcs and these are of the form R1(i) $\xrightarrow{co}$ W(i) $\xrightarrow{co}$ R2(i) (otherwise this is not the shortest cycle). Thus, R1, and R2 are from the same processor and so R1 $\xrightarrow{sco}$ R2.

(P7.2) By definition, an $\xrightarrow{sxo'}$ arc from X(i) to Y(j) on the cycle implies either X $\xrightarrow{spo}$ Y or X=W $\xrightarrow{co}$ R $\xrightarrow{spo'}$ Y, where $\xrightarrow{spo}$ and $\xrightarrow{spo'}$ are as defined by the aggressive conditions. (The $\xrightarrow{spo'}$ case occurs only for RCsc.)

The above properties imply that the cycle A(i) ... B(i) $\xrightarrow{co}$ A(i) implies a cycle on the operations of E_a that consists of $\xrightarrow{spo}$ or $\xrightarrow{spo'}$ arcs alternating with $\xrightarrow{sco}$ or $\xrightarrow{sco'}$ arcs. Furthermore, an $\xrightarrow{sco'}$ arc is always followed by an $\xrightarrow{spo'}$ arc. (Here $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{sco}$, and $\xrightarrow{sco'}$ are as defined by the aggressive conditions, and $\xrightarrow{sco'}$ and $\xrightarrow{spo'}$ occur only for RCsc.) Such a cycle implies a cycle of the form C ... D $\xrightarrow{co}$ C where the path from C to D is a multiprocessor dependence chain defined by the aggressive conditions. But then by the $\xrightarrow{sxo}$ condition, C(i) $\xrightarrow{xo}$ D(i) for all i. However, the cycle requires D(i) $\xrightarrow{xo}$ C(i) for all i, a contradiction. ■

Theorem 8: A system that obeys the conservative conditions for PC, RCsc, and RCpc also obeys the aggressive conditions for PC, RCsc, and RCpc respectively.

Proof: Let M represent any one of the models of PC, RCsc, and RCpc. Consider an execution, E_c, on a system that obeys the conservative conditions for M. Consider an $\xrightarrow{xo}$ of E_c that obeys the conservative conditions. The $\xrightarrow{xo}$ clearly obeys the initiation and termination parts of the aggressive conditions. It also obeys the uniprocessor dependence, coherence, and read-modify-write aspects of the $\xrightarrow{sxo}$ part of the aggressive conditions because of the $\xrightarrow{sxo}$ and $\xrightarrow{sxo''}$ parts of the conservative conditions. Thus, we only need to show that it also obeys the multiprocessor dependence chain part of the $\xrightarrow{sxo}$ part of the aggressive conditions. For this, we need to show that if there is a multiprocessor dependence chain from X to Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

We use the following properties that follow from the $\xrightarrow{sxo'}$ part of the conservative conditions and the definition of the various relations.

(P8.1) If A $\xrightarrow{spo}$ B, or A $\xrightarrow{spo'}$ B, or A $\xrightarrow{sco'}$ C $\xrightarrow{spo'}$ B, or A=W $\xrightarrow{sco}$ R $\xrightarrow{spo'}$ B (as defined by the aggressive conditions), then A(i) $\xrightarrow{xo}$ B(j) for all i,j except possibly if B is a write and Pj issues B.
(P8.2) If A $\xrightarrow{co}$ B or A $\xrightarrow{sco}$ B, then A(i) $\xrightarrow{xo}$ B(i) for all i.

The multiprocessor dependence chains of the aggressive conditions consist of $\xrightarrow{spo}$ or $\xrightarrow{spo'}$ arcs that alternate with $\xrightarrow{co}$, $\xrightarrow{sco}$, or $\xrightarrow{sco'}$ arcs (as defined by the aggressive conditions). The $\xrightarrow{sco'}$ arcs are always followed by $\xrightarrow{spo'}$ arcs. Further, the chains always start with $\xrightarrow{spo}$ or with W $\xrightarrow{sco}$ R $\xrightarrow{spo'}$, and always end with $\xrightarrow{spo}$ or $\xrightarrow{spo'}$. From the above two properties, it follows that for a multiprocessor dependence chain from X to Y, it must be that X(i) $\xrightarrow{xo}$ Y(j) for all i,j except possibly if Y is a write and Pj issues Y. When Y is a write and Pj issues Y, then by the $\xrightarrow{sxo}$ and $\xrightarrow{sxo''}$ parts of the conservative conditions, it follows that X(j) $\xrightarrow{xo}$ Y(j) whenever both X(j) and Y(j) are defined. Thus, it follows that X(i) $\xrightarrow{xo}$ Y(i) for all i and so the $\xrightarrow{sxo}$ part of the aggressive conditions is obeyed. ■

C.5: Proof for WO

This section shows that the aggressive conditions for WO (Figure 13) and the conservative conditions for this model (Figure 17) are equivalent.

Theorem 9 gives the proof that a system that obeys the aggressive conditions for WO also obeys the conservative conditions for that model. Theorem 10 gives the proof that shows that a system that obeys the conservative conditions for WO also obeys the aggressive conditions for that model.

Theorem 9: A system that obeys the aggressive conditions for WO also obeys the conservative conditions for WO.

Proof: The proof for this theorem is almost identical to the proof of theorem 7 given for RCsc. The main difference is that the conservative conditions for WO require W $\xrightarrow{sxo}$ R if W $\xrightarrow{po}$ R and W,R conflict, while the aggressive conditions do not require this. We can use the same technique used in Lemma 1.1 for SC whereby we can find an $\xrightarrow{xo}$ for any execution with the aggressive conditions of WO that obeys these conditions with a modification to the uniprocessor dependence part of $\xrightarrow{sxo}$ where if X and Y conflict and X $\xrightarrow{po}$ Y, then X $\xrightarrow{sxo}$ Y. Given this $\xrightarrow{xo}$, the proof of this theorem is identical to the proof of Theorem 7 for the RCsc model. ■

Theorem 10: A system that obeys the conservative condition for WO also obeys the aggressive conditions for WO.

Proof: The proof for this theorem is almost identical to the proof for Theorem 8 for the PC, RCsc, and RCpc models. ■

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of:

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ W

define $\xrightarrow{sxo}$, $\xrightarrow{sxo'}$, $\xrightarrow{sxo''}$:

X $\xrightarrow{sxo}$ Y if X and Y conflict and are the first and last operations in RW $\xrightarrow{po}$ W

X $\xrightarrow{sxo'}$ Y if X $\xrightarrow{spo}$ Y

X $\xrightarrow{sxo''}$ Y if X and Y conflict and are the first and last operations in one of

W $\xrightarrow{co}$ W

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo''}$ AR or AW $\xrightarrow{sxo''}$ W

Conditions on $\xrightarrow{xo}$:

If X $\xrightarrow{sxo}$ Y, X,Y by P_m, then X(m) $\xrightarrow{xo}$ Y(m).

If X $\xrightarrow{sxo'}$ Y, given X,Y by P_m, and one of

Y=R, then X(i) $\xrightarrow{xo}$ R(j) for all i,j, or

Y=W, then X(i) $\xrightarrow{xo}$ W(j) for all i,j except j=m.

If X $\xrightarrow{sxo''}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Initiation condition holds.

Termination condition holds for all sub-operations.

Figure 16: Conservative conditions for PC.

define $\xrightarrow{spo}$: X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of:

RWs $\xrightarrow{po}$ RWs

RW $\xrightarrow{po}$ RWs

RWs $\xrightarrow{po}$ RW

define $\xrightarrow{sxo}$, $\xrightarrow{sxo'}$, $\xrightarrow{sxo''}$:

X $\xrightarrow{sxo}$ Y if X and Y conflict and are the first and last operations in one of

RW $\xrightarrow{po}$ W

W $\xrightarrow{po}$ R

X $\xrightarrow{sxo'}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{spo}$ Y

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

R $\xrightarrow{rch}$ W

X $\xrightarrow{sxo''}$ Y if X and Y conflict and are the first and last operations in one of

W $\xrightarrow{co}$ W

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo''}$ AR or AW $\xrightarrow{sxo''}$ W

Conditions on $\xrightarrow{xo}$:

If X $\xrightarrow{sxo}$ Y, X,Y by P_m, then X(m) $\xrightarrow{xo}$ Y(m).

If X $\xrightarrow{sxo'}$ Y, then X(i) $\xrightarrow{xo}$ Y(j) for all i,j.

If X $\xrightarrow{sxo''}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Initiation condition holds.

Termination condition holds for all sub-operations.

Figure 17: Conservative conditions for WO.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$**:**

$X \xrightarrow{spo'} Y$ if X,Y are the first and last operations in $Xc \xrightarrow{po} Yc$

$X \xrightarrow{spo} Y$ if X,Y are the first and last operations in one of

$Rc\_acq \xrightarrow{po} RW$

$RW \xrightarrow{po} Wc\_rel$

define $\xrightarrow{sxo}$, $\xrightarrow{sxo'}$, $\xrightarrow{sxo''}$**:**

$X \xrightarrow{sxo} Y$ if X and Y conflict and are the first and last operations in $RW \xrightarrow{po} W$

$X \xrightarrow{sxo'} Y$ if X,Y are the first and last operations in one of

$X \xrightarrow{spo} Y$

$X \xrightarrow{spo'} Y$

$W \xrightarrow{co} R \xrightarrow{spo'} RW$

$R \xrightarrow{rch} W$

$X \xrightarrow{sxo''} Y$ if X and Y conflict and are the first and last operations in one of

$W \xrightarrow{co} W$

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either $W \xrightarrow{sxo''} AR$ or $AW \xrightarrow{sxo''} W$

Conditions on $\xrightarrow{xo}$**:**

If $X \xrightarrow{sxo} Y$, X,Y by P_m, then $X(m) \xrightarrow{xo} Y(m)$.

If $X \xrightarrow{sxo'} Y$, given X by P_m and Y by P_n, and one of

Y=R, then $X(i) \xrightarrow{xo} R(j)$ for all i,j, or

Y=W, then $X(i) \xrightarrow{xo} W(j)$ for all i,j except j=n.

If $X \xrightarrow{sxo''} Y$, then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Initiation condition holds.

Termination condition holds for all *competing* sub-operations.

Figure 18: Conservative conditions for RCsc.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$:

X $\xrightarrow{spo'}$ Y if X,Y are the first and last operations in one of

Rc $\xrightarrow{po}$ RWc

Wc $\xrightarrow{po}$ Wc

X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of

Rc_acq $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc_rel

define $\xrightarrow{sxo}$, $\xrightarrow{sxo'}$, $\xrightarrow{sxo''}$:

X $\xrightarrow{sxo}$ Y if X and Y conflict and are the first and last operations in RW $\xrightarrow{po}$ W

X $\xrightarrow{sxo'}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{spo}$ Y

X $\xrightarrow{spo'}$ Y

R $\xrightarrow{rch}$ W

X $\xrightarrow{sxo''}$ Y if X and Y conflict and are the first and last operations in one of

W $\xrightarrow{co}$ W

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then either W $\xrightarrow{sxo''}$ AR or AW $\xrightarrow{sxo''}$ W

Conditions on $\xrightarrow{xo}$:

If X $\xrightarrow{sxo}$ Y, X,Y by P_m, then X(m) $\xrightarrow{xo}$ Y(m).

If X $\xrightarrow{sxo'}$ Y, given X by P_m and Y by P_n, and one of

Y=R, then X(i) $\xrightarrow{xo}$ R(j) for all i,j, or

Y=W, then X(i) $\xrightarrow{xo}$ W(j) for all i,j except j=n.

If X $\xrightarrow{sxo''}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Initiation condition holds.

Termination condition holds for all *competing* sub-operations.

Figure 19: Conservative conditions for RCpc.

Appendix D: Aggressive Form of Uniprocessor Correctness Condition

This appendix discusses a relaxation of the uniprocessor correctness condition (Condition 1) given in Section 2.2. This condition is a straightforward adaptation of the condition developed for the PLpc model [AGG$^+$93]. As discussed in Section 2.2, the notion of a correct uniprocessor used in Condition 1 assumes the following: given an execution, E, of a correct uniprocessor, if an instruction instance, *i*, is in E, then the number of instruction instances in E that are ordered before *i* by program order is finite. Section 3 described how this restricts implementations by prohibiting the aggressive execution of operations that may be ordered after potentially non-terminating loops by program order. To allow aggressive implementations, we could eliminate the above constraint when defining uniprocessor correctness; i.e., we could assume that a uniprocessor can execute an instruction even if a loop before the instruction does not terminate in the execution.[30] Previous specifications of models have not explicitly stated which version of the condition is assumed; we therefore assumed the more conservative version in the text. However, we believe that except for SC, the intent of the remaining models can be captured even with the more aggressive uniprocessor correctness condition if one additional condition is enforced. We believe that the additional condition should ensure the following two conditions:

(1) For certain classes of well-behaved programs (e.g., DRF1/PL for RCsc or PLpc for RCpc), the models should appear SC (see Section 2.3.2).

(2) Consider any pair of shared-memory instructions X' and Y' from the same processor where X' is before Y' in the control flow graph of the processor. If the original specification of the model requires sub-operations of instances of X' to execute before those of Y', then the new condition should ensure that only a finite number of instances of X' are ordered by program order before any instance of Y'.

The additional condition is formalized below as Definition D.3 and is called the infinite execution condition. Parts (a), (b), and (c) of the definition cover (1) above (the proof for this is in [Adv93] along with a more formal set of conditions) and part (c) also covers (2) above. First we define some concepts that will be used by definition D.3.

Definition D.1: Loop
A *loop* in a control flow graph refers to a cycle in the control flow graph. A loop L *does not terminate* in an execution iff the number of instances of instructions from loop L in the execution is infinite.

Definition D.2: Infinite Execution Order ($\xrightarrow{ieo}$)
Let X and Y be two shared-memory instruction instances in an execution E, with X' and Y' being the corresponding memory operations.
If E is an execution on a WO (or PC) system, then X $\xrightarrow{ieo}$ Y iff X' $\xrightarrow{spo}$ Y', where $\xrightarrow{spo}$ is as defined for the conservative conditions of WO (or PC).
If E is an execution on a RCsc (or RCpc) system, then X $\xrightarrow{ieo}$ Y iff X' ($\xrightarrow{spo}$ | $\xrightarrow{spo'}$) Y', where $\xrightarrow{spo}$ and $\xrightarrow{spo'}$ are as defined for the conservative conditions of RCsc (or RCpc).
If E is an execution on a TSO system, then X $\xrightarrow{ieo}$ Y iff X=R $\xrightarrow{po}$ Y=RW or X=W $\xrightarrow{po}$ Y=W.
If E is an execution on a PSO system, then X $\xrightarrow{ieo}$ Y iff X=R $\xrightarrow{po}$ Y=RW or X=W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ Y=W.

Condition D.3: Infinite Execution Condition
Consider an execution E of program Prog that contains instruction instance j of instruction j'.
(a) If loop L does not terminate in some SC execution, then the number of instances of instructions in E that are from loop L and that are ordered by program order before *j* is finite.
(b) The number of instruction instances that are ordered before *j* by $\xrightarrow{rch'}$ in E is finite (for models that have $\xrightarrow{rch'}$ defined; e.g., WO, RCsc, RCpc).
(c) The number of instruction instances that are ordered before *j* by $\xrightarrow{ieo}$ in E is finite.

With the above modifications to our framework, a processor can execute a read (R) or write (W), even if it is not known whether previous loops (by program order) will terminate, as long as the loop is known to terminate in every SC execution and as long as no memory operation from the loop will be ordered before the R or W by

[30]This implies that $\xrightarrow{po}$ for a single processor no longer obeys the property of total orders described in footnote 5. We assume, however, that all other total orders discussed in this paper obey the property.

```
P1              P2
while (1);      if (A==1)
A = 1;              exit(0);
```

(a)

```
P1                          P2
while (A==0); (sync)        if (B==0)
B = 1;                          A = 1; (sync)
```

(b)

```
P1        P2                                P3
A = 1;    while (B==0);                     if (P==1)
B = 1;    if (A==0) {                           exit(0);
              while (1) { C = 1;} (sync)
              P = 1; (sync)
          }
```

(c)

Figure 20: Example code segments to illustrate the infinite execution condition.

$\xrightarrow{rch'}$ or by $\xrightarrow{ieo}$. Similarly, the compiler can move a shared-memory instruction i that is after a loop to before a loop that obeys the above conditions.

Figure 20 shows three example code segments to illustrate the use of the three parts of Condition D.3. Assume the memory model used is WO. Figure 20(a) shows a program that is properly labeled (PL). The write to A never occurs in any SC, and the exit statement is never executed under SC either. Furthermore, the while loop does not terminate in any SC execution. However, allowing the write to A to occur before the loop operations complete will lead to non-SC executions. Condition D.3(a) disallows this, however. Figure 20(b) shows an example where the while loop terminates in every SC execution. Again the program is properly labeled (PL) since the two competing operations are identified as *sync*. However, if we allow the write to B to occur before the loop completes, it is possible for P2 to see the new value of B and to therefore not do the write to A (again a non-SC result). The above is disallowed by Condition D.3(b). Finally, Figure 20(c) shows an example that is not properly labeled. P3 would never execute the exit instruction in any WO execution of this code. However, the exit instruction may get executed if we allow the write to P to occur on P2 before its preceding while loop ends. This is disallowed by Condition D.3(c).

Note that the above modification leads to optimizations only for loops that are known to terminate in all SC executions, and have shared-memory operations following such loops. The information about whether a loop will always terminate in an SC execution is often known to the programmer, and can be obtained from the programmer. In fact, most programs are written so that either they will terminate in all SC executions, or there are no shared-memory operations that follow a potentially non-terminating loop. Thus, the above modification is applicable to a large variety of programs.

Also note that the infinite execution condition for SC would imply that there is no gain in relaxing the uniprocessor correctness condition for SC.

The proofs of appendix C are applicable even with the above modification.