Weak Memory Consistency Models

Part I: Definitions and Comparisons[*][†]

Lisa Higham, Jalal Kawash, and Nathaly Verwaal
Department of Computer Science, The University of Calgary, Alberta, Canada T2N 1N4
email:<last name>@cpsc.ucalgary.ca

Abstract

Modern multiprocessors incorporate sophisticated memory structures in order to enhance performance. These structures allow processes to have inconsistent views of memory, which can, in turn, result in unexpected program outcomes. A memory consistency model is a set of guarantees that imposes constraints on the possible outcomes of sequences of interleaved and simultaneous operations in a multiprocessor. This paper presents a unifying framework to describe, compare and reason about memory consistency models. The framework is then used to give rigorous definitions of several widely used consistency models. These definitions are also shown to correspond to simple machine architectures. For each pair of models discussed, it is determined whether one is strictly stronger than the other or whether the two are incomparable.

1 Introduction

To enhance performance, multiprocessors incorporate sophisticated memory structures. These memories are either distributed-shared [31], replicate data through constructs such as caches and write buffers, or use advanced interconnection patterns including multiple buses. Any of these architectural features allow processes to have inconsistent views of memory, which, in turn, can result in unexpected program outcomes.

A memory consistency model is a set of guarantees describing constraints on the outcome of sequences of interleaved and simultaneous operations. Fewer guarantees allow more performance optimizations but yield complex machines that are difficult to program. Hence multiprocessor programmers need a precise description of the memory model of the underlying machine in order to construct correct and efficient programs.

Several memory consistency models have been described in the literature. Lamport's seminal work, beginning in 1978 [25, 26, 27], launched investigations into relaxations of sequential consistency, which was previously generally assumed. Dubois, Scheurich and Briggs were the first to actually propose a consistency model weaker than sequential consistency, namely weak ordering [9]. Their work differentiated between ordinary and synchronization operations. Synchronization operations are guaranteed to be sequentially consistent with respect to each other; however, ordinary operation may be re-ordered in a way that best suits performance. Later, the Stanford team built on weak ordering when developing the DASH multiprocessor with the memory model release consistency [12, 11]. Lipton and Sandberg defined and implemented a machine with quite different memory behavior, which they defined to be the Pipelined-RAM memory model [28]. Goodman, in 1989, was the first to explicitly state the notion of coherence or cache consistency [16], which is generally accepted as a minimum memory consistency requirement. He was also the first to define processor consistency. Currently, there are several distinct variants that use the same name [22, 33, 12, 5]. Herlihy and Wing defined the strongest memory consistency model, linearizability, which is often assumed by researchers in design and analysis of distributed algorithms [17].

[*]This research was supported by a trust fund and a post-graduate scholarship from the Natural Sciences and Engineering Research Council of Canada and by a research grant from The University of Calgary.

[†]Parts of this paper appeared an earlier conference paper [19].

A different *programmer-oriented approach* to the problem of non-sequentially-consistent memory was investigated independently at Stanford [12, 11] and Wisconsin-Madison [1, 2, 4], and lead to the notions of Properly Labeled (PL) and Data Race Free (DRF), respectively. PL constitutes a constrained style programming for a specific architecture. DRF develops a programming style and then suggests how to tailor the hardware to support it. These two teams later collaborated to combine their approaches [10].

Attiya, Chaudhuri, and Friedman have taken a complementary approach to DRF, which first specifies the memory consistency model and then develops a programming style [7].

Yet another approach is that of executable specifications for analyzing and verifying memory models. The formal verification laboratory at Stanford developed $Mur\phi$ as a description language and a verification system for finite state concurrent systems [8]. $Mur\phi$ was used to verify the SPARC v9 Relaxed Memory Order [30].

These memory consistency models arise from a wide variety of sources including architecture, system, and database designers, application programmers, and theoreticians. The descriptions of memory behavior use different types and degrees of formalism. Definitions range from precise and complicated axiomatic specifications to informal and sometimes ambiguous natural language descriptions. This makes the many different memory consistencies difficult to reason about or to compare. Programming for these models becomes inefficient when a new descriptive style must first be mastered for each change of model. A single unified formalization is needed that can specify any memory model addressed in the literature or provided by existing and future machines.

A few research groups have proposed such unifying frameworks to describe the operation of distributed shared memories. Gibbons and Merrit proposed an automata-based framework [14]. They start with a specific automaton to represent basic architectural assumptions. Then they define a memory consistency model as an automaton that is obtained by restricting the actions of the base automaton. They used their formalism to model release consistency and to prove that, as long as a program is properly labeled, release consistency is equivalent to sequential consistency [13]. At Xerox's Palo Alto Research Center, Sindhu, Frailong, and Cekleov [32] proposed an axiomatic framework that is based on three sets: memory operations, partial orders defined on memory operations, and axioms which are concerned with the legality of orders. A team at Georgia Tech developed another framework, which is based on partial orders [24, 5, 6].

This paper also presents a unifying framework that facilitates the description, analysis, and comparison of memory consistency models. Precision is essential for providing an unambiguous view of the logical behavior of a memory system. Unification provides a common basis with which memory consistency models can be compared. Our framework achieves both of these goals while remaining simple. It is based on the partial order ideas of Ahamad el al. [6] and has benefited from all the previous research just reviewed.

The framework is used to formally define several memory consistency models both from the literature and from existing machines. For each formal model [1], a corresponding distributed system architecture is described and proved equivalent to that model. Furthermore, the formalism is exploited to compare the various memory models and determine the relationship between each pair.

Our framework is further exploited in a companion paper [21] in which process coordination problems have been re-addressed in the context of weak memory consistency models.

The framework is presented in Section 2. Sections 3 and 4 are concerned respectively with pure memory models— those that do not make any differentiation between ordinary and synchronization operations and hybrid models— those that make this differentiation. Section 5 summarizes the contributions of the paper.

2 The Framework

Our aim is to provide a framework that is capable of describing the exact behavior of the memory of any distributed system of processes whether message-passing or shared-memory or any hybrid of the two. We model a multiprocessor system as a collection of processes operating on a collection of shared data objects, as is shown in figure 1.

One way to define a data object is to describe the object's initial state, the operations that can be applied to the object and the change of state that results from each applicable operation. For our model it suffices to

[1] With the exception of Goodman's Procssor Consistency which inherits features from two other memory models.

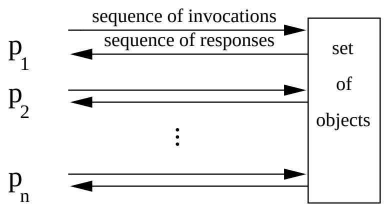

Figure 1: A multiprocessor system

define a data object to be the set of all sequences of allowable operations together with their results (similar to that of Herlihy and Wing [17]) as follows. An *action* is a 4-tuple (op, obj, in, out) where "op" is an operation, "obj" is an object name, and "in" and "out" are sequences of parameters. The action (op, obj, in, out) means that the operation "op" with input parameters "in" is applied to the object "obj" yielding the output parameters "out". A *(sequential data) object* is specified by a set of sequences of actions. A sequence of actions is *valid for object* x if and only if it is in the specification of x. For example, we specify a shared atomic read-write register by the set of all sequences $(o_1, o_2, ...)$ such that 1) each o_i is either a read action, denoted by a four-tuple $(read, x, \lambda, (v))$[2], that returns the value v of register x, or a write action, denoted $(write, x, (v), \lambda)$, that assigns a value v to register x, and 2) for every read action, the value returned is the same as the value written by the most recent preceding write action in the sequence.

An action o = (op, obj, in, out) can be decomposed into the two *matching* components, (op, obj, in), called the *action-invocation* and denoted invoc(o), and (op, obj, out), called the matching *action-response* and denoted resp(o). Let $(e_1, e_2, ...)$ be a sequence consisting of action-invocations and action-responses. Then e_j *follows* e_i if and only if $i < j$ and e_j *immediately follows* e_i if and only if $i = j - 1$.

A *timed-action* on an object is a 6-tuple (op, obj, in, invoc-time, out, resp-time) obtained by augmenting an action with the process's local time for the invocation and the response of the action. For a timed-action o, time(invoc(o)) (respectively, time(resp(o))) denotes the invoc-time (respectively, resp-time) of the timed-action.

Informally, a process interacts with data objects by issuing a stream of invocations to some subset of them and receiving a stream of responses that are interleaved with its invocations. This is formalized as follows. A *process* is a sequence of action-invocations. A *process execution* is a (possibly infinite) sequence of action-invocations and action-responses such that each response follows its matching invocation. A process execution is *blocking* if, for each action that has a non-empty output, the response of that action immediately follows its matching invocation.

Whether blocking or non-blocking, the natural notion of the computation of a process is the sequence of actions that arises as its invocations are processed. Therefore, a *process computation* is the sequence of actions created from a process execution by augmenting each invocation in the process sequence with its matching response.

A *(multiprocess) system*, (P, J), is a collection P of processes and a collection J of objects, such that each action-invocation of each process p in P is applied to an object in J. A *(multiprocess) system execution* for a system (P, J), is a collection of process executions, one for each p in P. Similarly, a *system computation* is a collection of process computations, one for each p in P.

Let (P, J) be a multiprocess system, and O be all the actions in a computation of this system. $O|p$ denotes all the actions o that are in the process computation of p in P. $O|x$ are all the actions that are applied to object x in J.

We define two partial orders[3] on the actions of a system. Action o_1 *program-precedes* o_2, denoted $o_1 \xrightarrow{prog} o_2$, if and only if invoc($o_2$) follows invoc($o_1$) in the definition of p (equivalently, o_2 follows o_1 in the computation of p). The partial order $(O, \xrightarrow{prog})$ is called the *program order*. Observe that for each process p in P, the program order is a total order on $O|p$. Let o_1 and o_2 be timed-actions in O. A timed-action o_1 *time-precedes* action o_2, denoted $o_1 \xrightarrow{time} o_2$, if and only if time(invoc(o_2)) $>$ time(resp(o_1)). The partial order $(O, \xrightarrow{time})$ is

[2]λ denotes the empty sequence.

[3]A partial order is a antisymmetric, transitive relation on a set. We denote a partial order by a pair (S, R) where S is a set and $R \subset S \times S$. The notation $s_1 R s_2$ means $(s_1, s_2) \in R$. When the set S is understood, R denotes the partial order. If $S' \subset S$ then (S', R) denotes the relation $(S, R \cap (S' \times S'))$.

called the *time order*.

For the definition of some memory consistency models it is necessary to distinguish the actions that change (write) a shared object from those that only inspect (read) a shared object. Let O_w denote that subset of O consisting of those actions in O that update a shared object, and O_r denote that subset consisting of the actions that only inspect a shared object. There are also some consistency models that provide other classes of actions. These are defined when needed.

Given any collection of actions O on a set of objects J, a *linearization of* O is a linear order[4] $(O, <_L)$ such that for each object x in J, the subsequence $(O|x, <_L)$ of $(O, <_L)$ is valid for x.

A *(memory) consistency condition* is a set of constraints on system computations. A computation C satisfies some consistency condition D if the computation meets all the constraints of D. A system provides memory consistency D if every computation that can arise from the system satisfies the consistency condition D.

The preceding definitions allow us to formalize memory consistency models in terms of constraints on computations. An additional goal is to associate common consistency conditions with machine architectures. A *(multiprocessor) machine* is a collection of processors together with various memory components. A machine *implements an action* by proceeding through a sequence of *events* that depend on the particular machine and that occur at the various components of the machine. A processor of a machine *implements a process* by implementing the actions corresponding to the action-invocations of the process, where these action implementations are initiated in program order. A multiprocessor machine *implements a system* (P, J) by having each processor implement a process in P. A *machine execution* is described by the sequence of resulting machine events.[5]

Each section in the remainder of this paper provides a definition of a memory consistency condition D and a description of a machine M. In each case we show that M and D correspond by establishing that every computation that can arise from an execution of M satisfies D and every computation that satisfies D could have been the result of an execution on M.

3 Pure Models

The literature describes many consistency conditions that may be considered "natural". The strongest, linearizability, is usually assumed by theoreticians designing distributed algorithms. One of the weakest, coherence, is typically assumed to be a necessary requirement of any reasonable machine. Numerous others fall between these two extremes and are often incomparable. We have chosen a selection of the most common to illustrate the appropriateness and flexibility of our framework.

In the following example computations, $w(x)v$ denotes a write action of value v to variable x. Similarly, $r(x)v$ denotes a read action from x returning v. We write the process identifier as a prefix for its own computation or computation prefix. The notation $w_p(x)v$ or $r_p(x)v$ is used to emphasize that these operations are performed by process p.

3.1 Sequential Consistency

Sequential consistency (henceforth abbreviated SC), defined by Lamport [26], is the most widely used memory consistency model. According to Lamport, a multiprocessor is said to be SC 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."

Several other papers [5, 17, 16, 29] describe SC; some use a different name or a different, but equivalent, definition.

Definition 3.1 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* SC *if there is a linearization $(O, <_L)$ such that $(O, \xrightarrow{prog}) \subseteq (O, <_L)$.*

[4] A linear order is an irreflexive partial order (S, R) such that $\forall x, y \in S\ x \neq y$, either xRy or yRx.

[5] Events in a multiprocessor can be simultaneous. For example, two different caches may be simultaneously updated. However, because the same outcome would arise if these simultaneous events were ordered one after the other in arbitrary order, we can assume that the outcome of a machine computation arises from a *sequence* of events.

Dubois, Scheurich and Briggs define *strong ordering* as a sufficient condition for SC [9] and Goodman states that “A system that adheres to this level of consistency is said to be a strongly ordered system” [16]. However, Adve and Hill show that strong ordering and SC are similar, but are not equivalent [3].

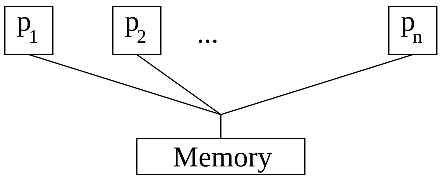

Figure 2: M_{SC}, a machine that implements SC

Figure 2 depicts a simple machine, M_{SC}, where each process is connected through bi-directional First-In-First-Out (bi-FIFO) channels to a switch, which is connected to memory via a single bi-FIFO channel. M_{SC} implements a read action (read, x, λ, v) by process p, with the ordered events 1) processor p sends a processor-read-request(x), and 2) memory sends a memory-read-reply(x, v). A write action (write, x, v, λ) is implemented with the ordered events 1) processor p sends a processor-write-request(x, v), and 2) memory performs a memory-update(x, v). For any computation C arising from an execution of M_{SC}, construct a sequence S of the actions of C by placing the actions in the order in which the corresponding *memory* events occurred. Sequence S must be in program order since the memory only services one action at a time, and the channels are FIFO. Moreover, it must be valid since each memory-read-reply will return the value of the last memory-update of the same cell. Thus S is a linearization that satisfies Definition 3.1. A simple argument in reverse can be used to show that any SC computation could have been executed on M_{SC}, thus establishing the following claim.

Claim 3.2 *M_{SC} implements exactly SC.*

An even stronger memory model, linearizability, was proposed by Herlihy and Wing [17]. For this model it is necessary to know the invocation and response time of each action. Mosberger calls linearizability *dynamic atomic consistency* [29]. An execution is linearizable if there is an assignment of each timed action o to one distinct point after time(invoc(o)) and before time(resp(o)) such that the resulting sequential computation is valid for each object. Herlihy and Wing require that each process’s execution is blocking; Definition 3.3 extends to non-blocking executions while agreeing with the definition of Herlihy and Wing when an execution is blocking.

Definition 3.3 *Let O be all the timed actions of a computation C of the multiprocess system (P, J). Then C is* linearizable *if there is a linearization $(O, <_L)$ satisfying: 1) $(O, \xrightarrow{prog}) \subseteq (O, <_L)$, and 2) $(O, \xrightarrow{time}) \subseteq (O, <_L)$.*

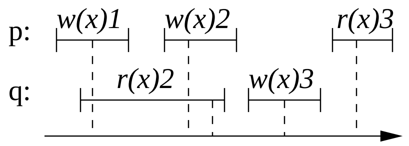

Figure 3: Linearizable sequence of actions

Figure 3 shows an execution of the system with $P = \{p, q\}$ and $J = \{x\}$ that is linearizable. Because the time interval of $r(x)2$ overlaps that of both $w(x)1$ and $w(x)2$ other total orderings may be considered but only the one indicated (by the projected dashed lines) is valid for object x (assuming $x \neq 2$ initially).

Figure 4 shows a computation that is SC, but not linearizable. It is SC because there is a valid ordering of all actions, which maintains program order. It is not linearizable, since no such ordering can also maintain time order.

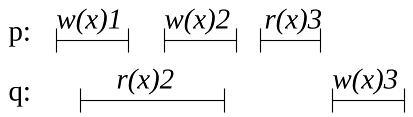

Figure 4: A SC but not linearizable computation

3.2 Coherence

Coherence, also called cache consistency [16], is among the weakest consistency conditions. Goodman states that coherence "only guarantees that accesses to a given memory location are strongly ordered" [16]. Mosberger indicates that "Coherence only requires that accesses are SC on a *per-location* basis" [29].

Definition 3.4 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* coherent *if for each object $x \in J$ there is some linearization $(O|x, <_{L_x})$ satisfying $(O|x, \xrightarrow{prog}) \subseteq (O|x, <_{L_x})$.*

Computation 1 is coherent but not SC. The linearizations for objects x and y are $<_{L_x} = w(x)0\ r(x)0\ w(x)1$ and $<_{L_y} = w(y)0\ r(y)0\ w(y)1$. However, there is no single linearization of all these actions that maintains program order.

Computation 1 $\left\{ \begin{array}{l} p : w(x)0\ w(x)1\ r(y)0 \\ q : w(y)0\ w(y)1\ r(x)0 \end{array} \right.$

Coherence has been described as a system where "all writes to the same location are serialized in some order and are performed in that order with respect to any processor" [12] (similarly in [2, 12, 24]). Furthermore, it is implicit in these informal descriptions that program order is maintained.

Definition 3.5 *Let O be all the actions of a computation C of some system (P, J). Then C is* coherent-var1 *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p, \xrightarrow{prog}) = (O|p, <_{L_p})$, and 2) $\forall q \in P$ and $\forall x \in J$ $(O_w|x, <_{L_p}) = (O_w|x, <_{L_q})$.*

Claim 3.6 *A computation is coherent if and only if it is coherent-var1 [33].*

Proof: Consider a coherent computation of a system (P, J). For each $p \in P$, we construct a sequence of actions ρ_p from the total order $(O|p, \xrightarrow{prog})$ and from all the linearizations $(O|x, <_{L_x})$ that satisfy Definition 3.4. Initially $\rho_p = (O|p, \xrightarrow{prog})$. Consider each $x \in J$ in turn. All actions, say $\{o_1, o_2, ..., o_k\}$, in $(O|p)|x$ appear in $(O|x, <_{L_x})$ in program order; hence they partition the sequence $(O|x, <_{L_x})$ into segments $S_0^x, S_1^x, ..., S_k^x$ such that $S_0^x, o_1, S_1^x, o_2, ..., o_k, S_k^x = (O|x, <_{L_x})$. Insert each sub-segment $S_{i_w}^x$ consisting of the sequence of writes in S_i^x into ρ_p anywhere between o_i and o_{i+1} (maintaining the order of $S_{i_w}^x$). In a similar way, we can show that ρ_p is a linearization for process p which satisfies the definition of coherent-var1.

Now consider a coherent-var1 computation of a system (P, J), and any $x \in J$. For each $p \in P$, let $(O|p \cup O_w, <_{L_p})$ be the linearization satisfying Definition 3.5. By part 2 of Definition 3.5, all processors have the same ordering of writes to x and by part 1 this ordering satisfies program order. Let ρ_x be this sequence of writes. Let r be any read of x by some process q. Suppose, in the linearization $((O|q \cup O_w)|x, <_{L_q})$, the closest write preceding r is $w(x)_i$, and the closest write succeeding r is $w(x)_{i+1}$. Note that the sequence of writes in $((O|q \cup O_w))|x, <_{L_p})$ is exactly ρ_x. Therefore, $w(x)_i$ and $w(x)_{i+1}$ are adjacent writes in ρ_x and r can be inserted between them while preserving validity and program order. In this way, every $r \in O_r|x$ can be inserted into ρ_x yielding a linearization of $O|x$ that preserves program order, and satisfies the requirements of Definition 3.4. ■

There is also a third possible definition of coherence, which arises because there are no restrictions on the ordering of actions to different objects.

Definition 3.7 *Let O be all the actions of a computation E of some system (P, J). Then E is* coherent-var2 *if there is some linearization $(O, <_L)$ such that $\forall x \in J$ $(O|x, \xrightarrow{prog}) \subseteq (O|x, <_L)$.*

Claim 3.8 *A computation is coherent if and only if it is coherent-var2 [33].*

Proof: For any coherent computation, create one sequence by placing the object linearizations, which satisfy Definition 3.4, one after the other. The result is a linearization that satisfies Definition 3.7 since it orders all the actions in the execution, such that program order on a per object basis is maintained. For any computation that satisfies Definition 3.7, and for each object x, set $(O|x, <_{L_x})$ equal to $(O|x, <_L)$. Then $(O|x, <_{L_x})$ is valid and maintains program order so it satisfies the requirements of Definition 3.4. ■

Since coherence, coherence-var1 and coherence-var2 are all equivalent, henceforth only the term coherence is used.

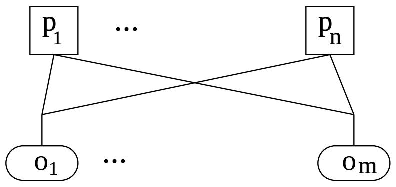

Figure 5: M_C, a machine that implements coherence

In the machine, M_C, in Figure 5, there is one switch for each object, and each processor is connected by a bi-FIFO channel to each switch. M_C implements a read action (read, x, λ, v) by process p, with the ordered events 1) a processor-read-request is placed on the channel connecting p to x, and 2) an object$_x$-read-reply(v) is placed on the channel from x to p. A write action (write, x, v, λ) by p is implemented with the ordered events 1) a processor-write-request(v) is placed on the channel connecting p to x, and 2) object$_x$-update(v) is performed by memory.

Claim 3.9 *The machine M_C, with non-blocking reads, implements exactly coherence.*

Proof: For any computation C arising from an execution of M_C, and for each object x, construct a sequence S_x of the actions of C on x by placing these actions in the order in which the corresponding *object* events occurred. It is easy to check that S_x is a linearization satisfying Definition 3.4.

Now, for any computation C that satisfies Definition 3.7, construct a sequence S of events that reflects how C could have arisen from an execution of M_C as follows. First, for each process p, construct a sequence S_p of processor-request events that corresponds to p. That is, the ith event of S_p is a processor-read-request (respectively, processor-write-request) placed on the channel from p to x, if and only if the ith action-invocation in process p is a read of (respectively, write to) object x. Also, from the linearization L that satisfies Definition 3.7 construct a sequence Q consisting of object-read-reply events and object-update events by setting the ith event in Q to be the object event that corresponds to the ith action in L. Now form S by concatenating the S_p's for each p (in any order) followed by Q. ■

3.3 Pipelined–RAM

Lipton and Sandberg [28] described the Pipelined Random Access Machine (P-RAM) with an architecture as shown in Figure 6. Each processor p has its own copy, μ_p, of the shared memory, and a FIFO channel connects every processor to every other processor's copy of memory. According to Lipton and Sandberg, read and write actions by process p are implemented in this machine as follows:

- Processor p implements a read(i) "by performing a normal read from location i" of μ_p.
- Processor p implements a write(i, v) "by performing a *local* action and initializing a *global* action. Locally, it does a normal write to $[\mu_p]$ at location i with value v. Globally, it sends a message $< i, v >$ to all the other processors."

They emphasize that, upon a write, a processor does not wait for the update to take effect in the copies of memory at other processors. What is not completely clear from this description, however, is whether or not reads and/or "local" writes are blocking; although the use of "performing" (as opposed to "initializing") seems to indicate a blocking activity. It is also unclear whether a processor first completes the update of

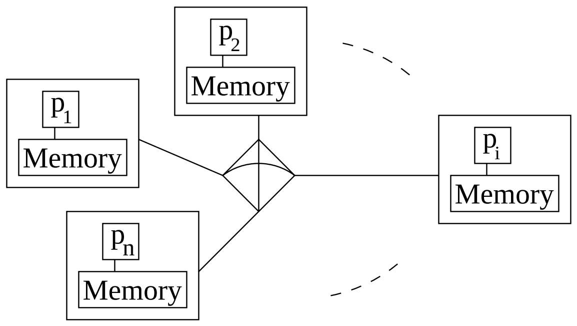

Figure 6: M_{PRAM}, a machine that implements P-RAM

its own copy of memory and then initiates the updates of other copies, or whether these events can happen in arbitrary order. Different memory models arise depending upon what assumptions are made concerning both of these issues.

Ahamed et al. [5] formalized one version of this machine, which, in our framework, becomes the following widely quoted definition of P-RAM:

Definition 3.10 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* P-RAM-A *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$.*

Any computation of any version of the machine in Figure 6 satifies the conditions of P-RAM-A. However, it is insufficient to describe the computation of any variant that is blocking on reads, or that has updates take effect at the writing processor's memory before they take effect at remote copies of memory.

Computation 2 $\begin{cases} p : w(y)0 \; r(y)1 \; w(x)1 \\ q : w(x)0 \; r(x)1 \; w(y)1 \end{cases}$

In Computation 2, process p observes q's write $w(y)1$, before it executes its own write $w(x)1$, and q observes p's write $w(x)1$ before it executes $w(y)1$. This requires that the reads be non-blocking. But notice that the linearizations $<_{L_p} = w_p(y)0 \; w_q(x)0 \; w_q(y)1 \; r_p(y)1 \; w_p(x)1$ and $<_{L_q} = w_q(x)0 \; w_p(y)0 \; w_p(x)1 \; r_q(x)1 \; w_q(y)1$ satisfy the conditions of P-RAM-A.

In contrast to Ahamed et al., Mosberger assumed that, upon a write-request, a processor first updates its own memory and subsequently broadcasts this update to all other processors [29]. In addition, after a read-request, a processor blocks until it receives the read-reply. Thus, if p observed a write, w_q, by some other process q before its own write w_p, then q must also observe w_q before w_p. That is, in the linearizations $(O|p \cup O_w, <_{L_p})$ that capture processes' views of the computation, $w_q <_{L_p} w_p \Rightarrow w_q <_{L_q} w_p$. Furthermore, if p observed that w_q occurred before w_p and some other process r observed w_p before its own write w_r, then q must also observe that w_q occurred before w_r. That is $w_q <_{L_p} w_p <_{L_r} w_r \Rightarrow w_q <_{L_q} w_r$. In general, the antecedent of this implication can be any finite length.

Definition 3.11 *Let O be all the actions of a computation C of the multiprocess system (P, J). Then C is* P-RAM *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) for any $m \geq 1$ and for all $w_{p_0}, w_{p_1}, \ldots, w_{p_m} \in O_w$, where w_{p_i} is a write by some process $p_i \in P$, if $w_{p_0} <_{L_{p_1}} w_{p_1} <_{L_{p_2}} w_{p_2} <_{L_{p_3}} \cdots <_{L_{p_m}} w_{p_m}$ then $w_{p_0} <_{L_{p_0}} w_{p_m}$.*

Let M_{PRAM} be the machine in Figure 6, together with the assumptions that reads are blocking and that any update occurs at the writer's copy of memory before it occurs at another processor's memory. In M_{PRAM}, a read action (read, x, λ, v) of p is implemented by processor p sending a processor-read-request(x) to μ_p, denoted p-read-request(x), and then blocking until it receives a memory-read-reply(v) from μ_p, denoted μ_p-read-reply(x,v). A write action (write, x, v, λ) by p is implemented with the ordered events (1) processor p sends processor-update-request(x, v) to μ_p, denoted p-update-request(x,v), (2) memory

μ_p applies a memory-update(x, v), (3) p sends a p-update-request(x,v) to μ_q for every $q \neq p$, and (4) for every $q \neq p$, in arbitrary order, memory μ_q applies memory-update(x, v), denoted by μ_q-update-by-$p(x,v)$.

Claim 3.12 *M_{PRAM} implements exactly P-RAM.*

Proof: Any execution E of M_{PRAM} can be described as a sequence (in time order) of events of the types: p-read-request(x), μ_p-read-reply(x,v), p-update-request(x,v), and μ_q-update-by-$p(x,v)$. Let $a \to b$ denote that event a precedes event b in E. The design of M_{PRAM} implies that the sequence E must satisfy the conditions: 1) p-read-request(x) immediately followed by μ_p-read-reply(x,v) for matching reads; 2) p-update-request$(x,v) \to \mu_q$-update-by-$p(x,v)$ for matching updates and for any p and q; 3) μ_p-update-by-$p(x,v) \to \mu_q$-update-by-$p(x,v)$ for matching memory updates; 4) the order of request events by p is the same as the order of the matching reply/update events at μ_p; 5) if p-update-request$(x,v) \to p$-update-request(y,w) then, for every q, the matching updates at μ_q, satisfy μ_q-update-by-$p(x,v) \to \mu_q$-update-by-$p(y,w)$. Recall that the sequence of request events for each processor p is the same as the order of the corresponding action-invocations in the process p.

From an execution E resulting in computation C, construct a subsequence for each process p by including only the "view of p's memory". That is, for each processor p, construct a sequence of the actions by p and all write actions in C in the same order in which the corresponding μ_p events occured in E. Then $<_{L_p}$ clearly satisfies program order (by conditions 4 and 5 of the sequence E) and consists of exactly $O|p \cup O_w$ (by construction) and is valid (by the property of memory). To show that the collection of sequences $<_{L_p}$ also satisfy condition 2 of P-RAM, assume that there exist $m \geq 1$, and $w_{p_0}, w_{p_1}, \ldots, w_{p_m} \in O_w$, where w_{p_i} is a write by process p_i such that $w_{p_0} <_{L_{p_1}} w_{p_1} <_{L_{p_2}} w_{p_2} <_{L_{p_3}} \cdots <_{L_{p_m}} w_{p_m}$. Then in E, μ_{p_1}-update-by-$p_0 \to \mu_{p_1}$-update-by-p_1 and μ_{p_2}-update-by-$p_1 \to \mu_{p_2}$-update-by-p_2 and $\ldots$ and μ_{p_m}-update-by-$p_{m-1} \to \mu_{p_m}$-update-by-p_m. By condition 3 of E, μ_{p_0}-update-by-$p_0 \to \mu_{p_1}$-update-by-$p_0 \to \mu_{p_1}$-update-by-$p_1 \to \mu_{p_2}$-update-by-$p_1 \to \mu_{p_2}$-update-by-$p_2 \to \ldots \to \mu_{p_m}$-update-by-$p_{m-1} \to \mu_{p_m}$-update-by-$p_m \to \mu_{p_0}$-update-by-$p_m$. That is μ_{p_0}-update-by-$p_0 \to \mu_{p_0}$-update-by-p_m. Hence, by our construction, $w_{p_0} <_{L_{p_0}} w_{p_m}$.

Now consider any computation, C, that satisfies Definition 3.11. Construct a corresponding sequence of events E for M_{PRAM} from an n-way merge of the n linearizations $(O|p \cup O_w, <_{L_p}) = o_1^p, o_2^p, \ldots, o_k^p$ as follows. Initially, $E = \lambda$ and $L_p = (O|p \cup O_w, <_{L_p})$ for each p. Consider the first remaining action o_i^p of each sequence L_p in turn (the first time this will be action o_1^p for each process p). If $o_i^p \in O|p$, append to E the corresponding p and μ_p events in that order and remove o_i^p from $<_{L_p}$. If the corresponding memory-update μ_q-update-by-q is already in E, then append μ_p-update-by-q to E, and remove the action o_i^p from L_p. Otherwise, $o_i^p \in O|q \cap O_w$, leave L_p unchanged and consider the first remaining action of the next sequence $L_{p'}$. It is straightforward to check that the sequence E so constructed satisfies conditions 1 through 5, provided that this construction exhausts each L_p.

So it remains to verify that all events associated with the execution are necessarily added to E. Assume instead that at some point in the construction of E, each sequence L_p begins with a write $o_i^p \in O_w|q$ for some $q \neq p$ and the corresponding processor-update-request is not yet in E. Then there must be a cyclic sequence $p_{\alpha_1}, p_{\alpha_2}, \ldots p_{\alpha_k}$ of processes such that $L_{p\alpha_i}$ begins with a write by process $p_{\alpha_{i+1}}$ labeled $w_{\alpha_{i+1}}$ and L_{p_k} begins with a write by process p_{α_1}. Also, w_{α_i} must be in L_{α_i} since otherwise the construction could proceed. But this contradicts condition 2 of P-RAM. Hence the construction of E completes, and E describes an execution on M_{PRAM} of C. ■

We complete this section with comparisons between P-RAM, P-RAM-A and the other consistency conditions previously defined.

Coherence and P-RAM-A are non-comparable. Computation 1 is P-RAM-A as well as coherent.

Computation 3 $\left\{ \begin{array}{l} p : w(x)0 \ r(x)1 \\ q : w(x)1 \ r(x)0 \end{array} \right.$

The linearizations $<_{L_p} = w(x)0 \ w(x)1 \ r(x)1$ and $<_{L_q} = w(x)1 \ w(x)0 \ r(x)0$ show that Computation 3 is P-RAM-A since they maintain program order. However, it is not coherent because is not possible to construct a linearization of all the actions to location x that maintains program order.

Computation 4 $\left\{ \begin{array}{l} p : w(x)0 \ w(x)1 \ w(y)2 \\ q : r(y)2 \ r(x)0 \end{array} \right.$

In Computation 4, let the linearization for the objects be: $<_{L_x} = w(x)0\ r(x)0\ w(x)1$ and $<_{L_y} = w(y)2\ r(y)2$. Both these linearizations maintain program order and thus the computation is coherent. Since it is not possible to construct a linearization for the actions by q together with the writes by p that extends program order, the computation is not P-RAM-A.

Finally, any execution that is P-RAM is also P-RAM-A. Computation 3 is coherent but not P-RAM-A (and thus not P-RAM) and Computation 4 is P-RAM but not coherent. Hence, P-RAM and coherence are non-comparable.

3.4 Processor Consistency

The term processor consistency was first used by Goodman [16] to capture a consistency condition that is stronger than coherence but weaker than SC. Many others [5, 12, 24, 29, 11] have used the same term to define memory consistency models that have in common Goodman's original intentions, but that differ in subtle ways. A paper that reveals the relations and differences between these different versions of processor consistency is in preparation [22]. Here, our discussion is limited to the original definition by Goodman (henceforth PC-G). In addition to coherence, Goodman[6] required that [16]: " the order in which writes from two processes occur, as observed by themselves or a third process need not be identical, but writes issuing from any process may not be observed in any order other than that in which they are issued."

Goodman allows the interleaving of writes by two different processes to be viewed differently by each process, as long as program order and coherence is maintained. The following definition is based in the interpretation of Ahamed et al. [5].

Definition 3.13 *Let O be all the actions of a computation C of a multiprocess system (P, J). Then C is* PC-G *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) $\forall q \in P$ and $\forall x \in J$ $(O_w|x, <_{L_p}) = (O_w|x, <_{L_q})$.*

By comparing Definition 3.13 with Definition 3.5 it is easily confirmed that PC-G implies coherence [33]. Computations 2 and 5 illustrate that PC-G and P-RAM are not comparable. Computation 2 is not P-RAM; however, the linearizations $<_{L_p} = w_p(y)0\ w_q(x)0\ w_q(y)1\ r_p(y)1\ w_p(x)1$, and $<_{L_q} = w_q(x)0\ w_p(y)0\ w_p(x)1\ r_q(x)1\ w_q(y)1$ satisfy the PC-G conditions.

Computation 5 $\begin{cases} p: w(x)1\ w(y)1\ r(y)0 \\ q: w(y)0\ w(x)0\ r(x)1 \end{cases}$

Computation 5 is not PC-G because it is not possible to build the required linearizations for p and q that preserve program-order and agree on the ordering of write actions to the same location. The linearizations $<_{L_p} = w_p(x)1\ w_p(y)1\ w_q(y)0\ r_p(y)0\ w_q(x)0$ and $<_{L_q} = w_q(y)0\ w_q(x)0\ w_p(x)1\ r_q(x)1\ w_p(y)1$ however, show that Computation 5 is P-RAM.

The first condition of PC-G is exactly the definition of P-RAM-A. Furthermore, since Computation 5 is P-RAM, PC-G is strictly stronger than P-RAM-A. Although PC-G implies both P-RAM-A and coherence, an execution that is both coherent and P-RAM-A is not necessarily PC-G, contrary to some previous claims (see [23] for example). Computation 5 illustrates this since the linearizations $<_{L_x} = w_q(x)0\ w_p(x)1\ r_q(x)1$ and $<_{L_y} = w_p(y)1\ w_q(y)0\ r_p(y)0$ satisfy coherence. Finally, computation 1 is PC-G, establishing that PC-G is strictly weaker than SC.

4 Hybrid Models

Hybrid models classify operations as special or ordinary according to their function. Further classification has also been proposed [10]. These models aim at utilizing system optimizations while still appearing SC to the programmer. In this section, we describe three hybrid models: weak ordering, SPARC's total store ordering and partial store ordering. More hybrid models, such as release consistency [12, 11] and Java consistency [18, 15], have been proposed and can also be described using our framework.

[6] Goodman uses the term weak ordering instead of coherence. In the literature, weak ordering usually refers to a different consistency model. See Section 4.1.

The formal definitions of some hybrid models might seem overly complex, especially compared to more informal definitions. However, the subtle problems that a programmer encounters in systems that implement a hybrid memory model are underlined by these formal definitions, and obscured by more informal descriptions.

4.1 Weak Ordering

Dubois, Scheurich and Briggs were the first to propose weak ordering (WO) [9] (Gharachorloo et al. call it weak consistency [12]). A WO system distinguishes between ordinary and synchronization actions, and guarantees minimum constraints for each class. The synchronization actions must satisfy SC, and all processes must view an ordinary action before (respectively after) a synchronization action if program order places it before (respectively after) the synchronization action. Dubois, Scheurich and Briggs also state that actions to the same object must remain in program order.

In a WO system, program order for ordinary actions can sometimes be violated. Define *weak program order*, denoted $\overset{weak-prog}{\longrightarrow}$, by $o_1 \overset{weak-prog}{\longrightarrow} o_2$ if $o_1 \overset{prog}{\longrightarrow} o_2$ and either 1) at least one of $\{o_1, o_2\}$ is a synchronization action, or 2) $\exists o'$ such that o' is a synchronization action and $o_1 \overset{prog}{\longrightarrow} o' \overset{prog}{\longrightarrow} o_2$, or 3) o_1 and o_2 are to the same object.

Definition 4.1 *Let O be all the actions of a computation C of a multiprocess system (P, J). Then C is* WO *if for each process $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \overset{weak-prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) $\forall q \in P$ $(O_w \cap O_{synch}, <_{L_p}) = (O_w \cap O_{synch}, <_{L_q})$.*

Without any synchronization actions, WO is weaker even than P-RAM. Condition 1 of Definition 4.1 only guarantees weak program order, whereas P-RAM computations must maintain program order. Weak ordering is also weaker than coherence. Even though WO ensures that program order on a per object basis is maintained, not all processes necessarily see writes to the same object in the same order, as is required in coherent systems. Dubois, Scheurich and Briggs however, seem to imply that a system should be coherent and satisfy the conditions of WO simultaneously [9]. The following defines a weakly-ordered-coherent (WO$_{coherent}$). Note that, WO$_{coherent}$ system is equivalent to coherence if no synchronization is used.

Definition 4.2 *Let O be all the actions of a computation C of a multiprocess system (P, J). Then C is* WO$_{coherent}$ *if for each process $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ satisfying the two conditions of Definition 4.1 and $\forall q \in P$ and $\forall x \in J$ $(O_w|x, <_{L_p}) = (O_w|x, <_{L_q})$.*

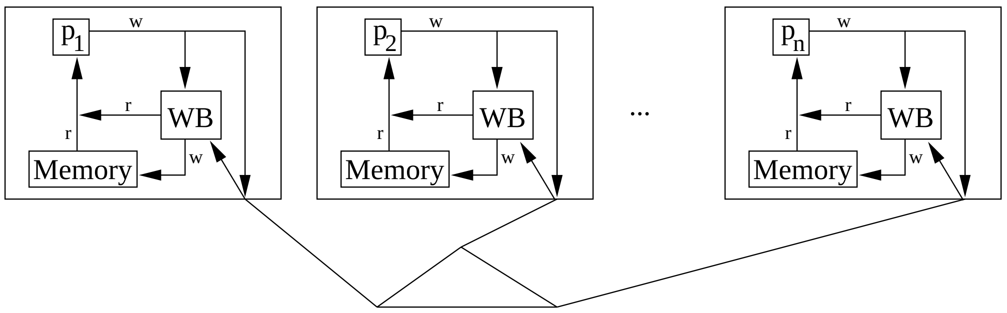

Figure 7: a machine that implements WO

The machine in Figure 7 has n processors where each has its own copy of the shared memory and a write buffer. These processors are connected by a complete network of FIFO channels. A write is placed in the write buffer and sent to every other processor in the system. A write received from another processor is also placed in the write buffer. The write buffer sends the writes to the memory, but FIFO order is only guaranteed on a per object basis. A read first checks the write buffer. If there is a write to the same object, the value of the last such write is sent back to the processor, otherwise the read is sent to the memory.

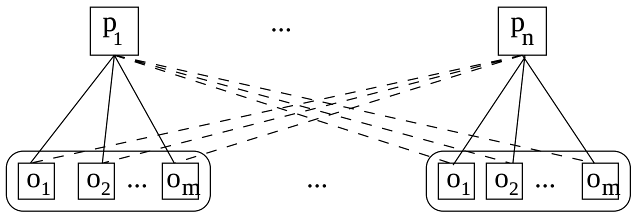

Figure 8: M_{WO}, a machine that implements WO

We could also view the write buffer and the memory together as one unit. Since the write buffer is only guaranteed to empty in FIFO manner on a per object basis, it is as if there is a separate FIFO channel for each object in the memory as depicted in Figure 8. In M_{WO}, a processor p has a bi-FIFO read-write channel (represented by solid lines) to each object in its own copy of memory, μ_p. Furthermore, each processor p has a FIFO write channel (represented by dashed lines) to each object in each other copy of μ_q of memory, where $p \neq q$. In M_{WO}, a read action (read, x, λ, v) by p is implemented by the ordered sequence of events (1) processor p sends a processor-read-request to x in μ_p and (2) object x of μ_p sends the matching μ_p-object$_x$-read-reply(v) to p. A write action (write, x, v, λ) by p is implemented by the ordered events (1) processor p sends processor-update-request(v) to each copy of object x (2) each process q's copy of object x applies the matching μ_q-object$_x$-update(v).

Claim 4.3 *M_{WO} implements exactly WO.*

Proof: For any computation C arising from an execution of M_{WO}, and for each processor p, construct a sequence S_p of the actions of C by p and all writes by placing these actions in the order in which the corresponding μ_p events occurred. Since each object services only one request at a time, and the channels are FIFO, S_p must satisfy program order on a per object basis. Also, S_p must be valid since each μ_p-object$_x$-read-reply will return the value of the last μ_p-object$_x$-update. Thus for each p, S_p is a linearization that satisfies Definition 4.1.

Now consider any computation that satisfies Definition 4.1. Construct a corresponding sequence of events E for M_{WO} from an n-way merge of the n linearizations $(O|p \cup O_w, <_{L_p}) = o_1^p, o_2^p, \ldots, o_k^p$ as follows. Initially, $E = \lambda$, $L_p = (O|p \cup O_w, <_{L_p})$ for each p and $P_p = (O|p, \xrightarrow{prog})$ for each p. Consider the first remaining action $o_i^p \in (O|q \cap O|x)$ for some object x and some process q of each sequence L_p in turn (the first time this will be action o_1^p for each process p). If the processor event corresponding to o_i^p is already in E then append to E the μ_p event corresponding to the action o_i^p and remove o_i^p from L_p. If the processor event corresponding to o_i^p is not in E yet then consider P_q. Append the processor events corresponding with each action ordered before o_i^p in P_q in the same order as the corresponding actions are ordered in P_q, and remove all these actions from P_q. Now append to E the processor and μ_p events corresponding to the action o_i^p and remove o_i^p from P_q and L_p.

All events associated with the computation C are obviously added to E. By construction, all processor events are in the same order as the action invocations of each process. When considering the events at μ_p's copy of some object x, these must be in processor order, since the linearization $(O|p \cup O_w, <_{L_p})$ is in program order with respect to object x, and could thus have been send along the FIFO channels connected to μ_p's copy of x. And thus E could have been executed on M_{WO}. ■

4.2 SPARC

Figure 4.2 shows the SPARC [34] architecture. There is one store buffer associated with each processor in the system and a main memory, which is single ported with a non-deterministic switch providing one memory access at a time. Three actions that access memory are supported: write, read, and swap-atomic. A write action (write, l, v, λ) by process p is implemented by 1) a p-buf-str(l, v) event adds (l, v) to processor p's store buffer; 2) a matching p-mem-update(l, v) event commits the pending store to main memory. Writes are non-blocking; p can invoke its next action as soon as the p-buf-str(l, v) event completes. A read action

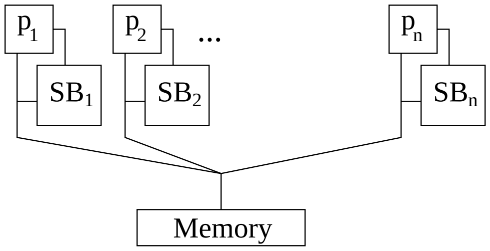

Figure 9: The SPARC machine

(read, l, λ, v) by process p is implemented by 1) a p-read-req(l) is sent by processor p to its write buffer; 2) If there is a store to location l pending in p's store buffer, then the matching p-read-reply(l,v) event returns to p the value v of the latest such store to l. Otherwise, the matching p-read-reply(l,v) event returns to p the value of l in main memory. Reads are blocking; p cannot invoke its next action until the matching p-read-reply(l) completes. A swap-atomic action (sa, l, v, w) by process p has both read and write semantics and is implemented as the single event p-swap(l,v,w), which has the same effect as the indivisible concatenation of p-buf-str(l,v), p-mem-reply(l,w), and p-mem-update(l,v)[7]. The swap-atomic is blocking.

The SPARC [34] architecture implements, selectively, Total Store Ordering (TSO) and Partial Store Ordering (PSO). The order in which p-mem-update(,) events commit values to main memory differentiates between them. If the buffers behave as FIFO queues, the model is called TSO. The model is PSO, if it is only guaranteed that stores performed to the same location are sent to main memory in FIFO order. The FIFO queue and the atomicity of the swap-atomic together imply that, for TSO, swap-atomic flushes the entire store buffer.

Let M_{TSO} denote the SPARC under TSO control. An execution of M_{TSO} can be described as a sequence of the types of events previously described. Because reads are blocking, however, any execution will produce the same computation as one in which each p-read-req(l) is moved to immediately before its matching p-read-reply(l,v). These reads can then be combined to one event denoted, p-read(l,v). Call any such execution with this form a *collapsed* execution. Let $a \to b$ denote that a precedes b in sequence E. Any collapsed execution E must satisfy the conditions: 1) p-buf-str$(l,v) \to p$-mem-update(l,v) for matching events; 2) if p-buf-str$(l,v) \to p$-buf-str(h,w) then p-mem-update$(l,v) \to p$-mem-update(h,w). 3) For a given event $e = p$-read(l,v), let $\hat{e}$ be the most recent q-mem-update(l,w) event that precedes e. If there is no p-buf-str(l,x) between $\hat{e}$ and e, then $v = w$. Otherwise, $v = x$ where p-buf-str(l,x) is the latest such event between $\hat{e}$ and e. 4) A p-swap(l,v,w) can be replaced by the indivisible concatenation of p-read(l,w), p-buf-str(l,v), p-mem-update(l,v) and properties 1 through 3 must still hold.

Some observations facilitate modelling the execution of M_{TSO} by constraints on computations. Besides it own events, each processor can "see" only another processor q's q-mem-update(,) events. A store by p is visible to p (as p-buf-str(,)) before it is visible to other processors. If p-buf-str$(l,v) \to q$-mem-update(l,v') $\to p$-mem-update(l,v), then the write by q to l is invisible to p.

In the following, $(A \uplus B)$ denotes the disjoint union of A and B, and if $x \in A \cap B$ then the copy of x in A is denoted x_A and the copy of x in B is denoted x_B. Let O_a denote all the swap atomic actions; O_w denote all actions that update objects (writes and swap-atomics) and O_r denote all actions that inspect objects (reads and swap-atomics).

Definition 4.4 *Let O be all the actions of a computation C of the multiprocess system (P,J). Then C is TSO if there exists a total order $(O_w, \xrightarrow{writes})$ such that $(O_w, \xrightarrow{prog}) \subseteq (O_w, \xrightarrow{writes})$ and $\forall p \in P$ there is a total order $(O|p \uplus O_w, \xrightarrow{merge_p})$, satisfying:*

1. *$(O|p, \xrightarrow{prog}) = (O|p, \xrightarrow{merge_p})$, and*

2. *$(O_w, \xrightarrow{writes}) = (O_w, \xrightarrow{merge_p})$, and*

[7]Actually, the swap-atomic may proceed concurrently with other events by other processors, and hence not be strictly indivisible, but the outcome of any execution of a swap-atomic is identical to one in which the three component pieces happen without any intervening events.

3. *if $w \in (O|p \cap O_w)$ then $w_{O|p} \overset{merge_p}{\longrightarrow} w_{O_w}$, and*

4. *$((O|p \uplus O_w) \backslash (O_{invisible_p} \cup O_{memwrites_p}), \overset{merge_p}{\longrightarrow})$ is a linearization, where*
   $O_{invisible_p} = \{w \mid w \in (O_w \backslash O|p) \cap O|x \wedge \exists w' \in O|x \cap O|p \cap O_w \wedge w'_{O|p} \overset{merge_p}{\longrightarrow} w \overset{merge_p}{\longrightarrow} w'_{O_w}\}$
   $O_{memwrites_p} = \{w_{O_w} \mid w \in O|p \cap O_w\}$, and

5. *let $w \in (O|p \cap O_w)$ and $a \in (O|p \cap O_a)$, if $w \overset{prog}{\longrightarrow} a$, then $w_{O_w} \overset{merge_p}{\longrightarrow} a$, and if $a \overset{prog}{\longrightarrow} w$, then $a \overset{merge_p}{\longrightarrow} w_{O|p}$*

Theorem 4.5 *M_{TSO} implements exactly TSO.*

Proof: Let E be a collapsed execution on M_{TSO} of some multiprocessor system (P, J). For any subsequence $\hat{E}$ of E, let $S(\hat{E})$ denote the sequence obtained from E by replacing each p-read(l, v) by p's corresponding action (read,l,λ,v); each p-buf-str(l, v) or p-mem-update(l, v) by p's corresponding action (write,l,v,λ); and each p-swap(l, v, w) by p's corresponding action (sa,l,v,w). Let E_{writes} be the subsequence of E containing all q-mem-update(,) and q-swap(,) events for all $q \in P$. By condition 2 for E, $S(E_{writes})$ is a total order on O_w satisfying program order. For each process $p \in P$, let E_p be the subsequence of E containing E_{writes} together with all p-buf-str(,) and p-read(,) events. Form E'_p from E_p by duplicating each p-swap(,). Then $S(E'_p)$ is a total order on $O|p \uplus O_w$, where $O|p$ is the set of actions corresponding to the p-buf-str(,) and p-read(,) events, and the first copy of each p-swap(,) event; and O_w is the set of actions corresponding to those in E_{writes}. It is easily checked that $S(E'_p)$ satisfies properties 1, 2, 3, and 5 of TSO. To establish property 4 of TSO, notice that the actions in $O_{memwrites_p}$ correspond to the events p-mem-update(,) and the second copy of each p-swap(,); and those in $O_{invisible_p}$ correspond to those q-mem-update(,) events satisfying p-buf-str$(l,) \rightarrow q$-mem-update$(l,) \rightarrow p$-mem-update$(l,)$ and $q \neq p$. By condition 3, removing from $O|p \uplus O_w$ the actions corresponding to these events, leaves a sequence that is valid for every object. Thus $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memwrites_p}), \overset{merge_p}{\longrightarrow})$ is a linearization.

Let C be a computation satisfying TSO. From the sequences $(O|p \uplus O_w, \overset{merge_p}{\longrightarrow})$ construct sequences E_p by replacing each (read,l, λ, v) in $O|p$ with the event p-read(l, v); each (write,l, v, λ) in $O|p$ (respectively, O_w) with p-buf-str(l, v) (respectively, p-mem-update(l, v)). Finally, remove each (sa,l, v, w) in $O|p$ and replace each (sa,l, v, w) in O_w with p-swap(l, v, w). Property 2 and 5 of TSO, ensures that it is possible to form a sequence E by merging the E_p while identifying all the matching p-mem-update(,) and p-swap(,) events for every p. Combine the segments of each E_p that are between the ith and $i + 1$st p-mem-update(,) events with an arbitrary merge. Properties 1, 2, and 3 ensure that any such merge will satisfy the conditions 1, 2, and 4 required of E. Condition 3 follows from property 4 of TSO. ■

Denote the SPARC machine with PSO control by M_{PSO}. M_{PSO} provides fewer guarantees than the M_{TSO}; specifically, condition 2 of executions on M_{TSO} is weakened to apply only to update events on the same object. Therefore, a swap-atomic action will only flush those writes that are to the same location. For this reason, PSO also provides a store-barrier instruction, which does not access memory but imposes additional constraints on the order in which memory updates leave the store buffer. Specifically, no store invoked after a store-barrier is allowed to complete before any store that is invoked before the store-barrier.

A (collapsed) execution E of M_{PSO} can, therefore, be described as a sequence of events of the types p-read(l), p-buf-str(l, v), p-mem-update(l, v), p-swap(l, v, w), and p-barrier. Execution E must satisfy the conditions 1, 3, and 4 of M_{TSO} and the condition 2' given by: 2') if p-buf-str$(l, v) \rightarrow p$-buf-str(l, w) then p-mem-update$(l, v) \rightarrow p$-mem-update(l, w). Furthermore, the p-barrier events impose the condition: 5) if p-buf-str$(l, v) \rightarrow p$-barrier $\rightarrow p$-buf-str(h, w) then p-mem-update$(l, v) \rightarrow p$-mem-update(h, w).

Let O_{sb} denote all the store barrier actions in O.

Definition 4.6 *Let O be all the actions resulting from an execution E of the multiprocess system (P, J). Then E is PSO if there exists a total order $(O_w, \overset{writes}{\longrightarrow})$ such that $\forall x$, $(O_w \cap O|x, \overset{prog}{\longrightarrow}) \subseteq (O_w \cap O|x, \overset{writes}{\longrightarrow})$ and $\forall p \in P$ there is a total order $(O|p \uplus O_w, \overset{merge_p}{\longrightarrow})$, satisfying items 1 through 4 of TSO and (5) if $sb \in (O|p \cap O_{sb})$ and w, $u \in (O|p \cap O_w)$ and $w \overset{prog}{\longrightarrow} sb$ and $sb \overset{prog}{\longrightarrow} u$, then $w_{O_w} \overset{merge_p}{\longrightarrow} u_{O_w}$.*

The proof that the PSO consistency condition corresponds to the SPARC machine under PSO control is similar to the proof of Theorem 4.5; details can be found elsewhere [20].

Theorem 4.7 *M_{PSO} implements exactly PSO.*

Clearly, any TSO execution is PSO. Computation 4 is PSO as shown by the writes order $w_p(y)2 \xrightarrow{writes} w_p(x)0 \xrightarrow{writes} w_p(x)1$. But it is not TSO because the required linearization for q does not exist. Hence, TSO is strictly stronger than PSO.

To compare TSO and PSO with the other models we assume $O_a = \emptyset$.

Computation 6 $\begin{cases} p : w(x)0\ r(y)1\ r(y)2\ r(y)3\ r(x)0 \\ q : w(y)1\ w(y)2\ w(x)6\ w(y)3 \end{cases}$

It can be shown [20] that Computation 6 is TSO but not P-RAM-A. Hence TSO (or PSO) does not imply P-RAM-A or P-RAM or PC-G or SC. Any TSO (PSO) execution is coherent however [20].

Computation 7 $\begin{cases} p : w(x)1 \\ q : w(y)1 \\ r : w(x)0\ r(x)1\ r(y)0 \\ s : w(y)0\ r(y)1\ r(x)0 \end{cases}$

Finally, Computation 7 is coherent and PC-G but not PSO [20]. Thus neither coherence nor PC-G implies even PSO.

5 Conclusion

We have presented a unifying formal framework to describe memory consistency models. This formalism was exploited to reveal substantial differences between models. We further exploit this formalism to study process coordination problems in the context of weak memory models in a companion paper [21].

When no explicit synchronization primitives are used, the models described yield the relationships depicted in Figure 10. An arrow from some memory consistency model A to another memory model B, indicates that any system satisfying the constraints of model A will also satisfy the constraints of model B.

Table 1 summarizes the relationships between computations and models presented in this paper.

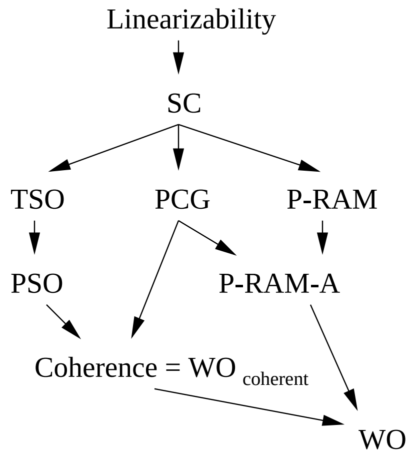

Figure 10: Relationships between memory consistency models

References

[1] S. V. Adve. Using information from the programmer to implement system optimizations without violating sequential consistency. Technical Report ECE 9603, Department of Electrical and Computer Engineering, Rice University, March 1996.

Table 1: Model-computation relationships

	linearizability	SC	coherence	P-RAM-A	P-RAM	PC-G	WO	TSO	PSO
Figure 3	√	√	√	√	√	√	√	√	√
Figure 4		√	√	√	√	√	√	√	√
Computation 1			√	√	√	√	√	√	√
Computation 3				√	√		√		
Computation 4			√				√		√
Computation 2			√	√		√	√		
Computation 5			√	√	√		√		√
Computation 6			√				√	√	√
Computation 7			√	√	√	√	√		

[2] S. V. Adve and M. D. Hill. Implementing sequential consistency in cache-based systems. *1990 Int'l Conf. on Parallel Processing*, pages I47–I50, August 1990.

[3] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, pages 2–14, May 1990.

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

[5] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Also available as College of Computing, Georgia Institute of Technology technical report GIT-CC-92/34.

[6] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementations, and programming. *Distributed Computing*, 9:37–49, 1995.

[7] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 241–250, 1993.

[8] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specifications of abstract memory models. In *Proc. of the 1993 Int'l Symp. on Research on Integrated Systems*, March 1993.

[9] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *Proc. of the 13th Annual Int'l Symp. on Computer Architecture*, pages 434–442, June 1986.

[10] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy. Revision to memory consistency and event ordering in scalable shared-memory multiprocessors. Technical Report CSL-TR-93-568, Computer Systems Laboratory, Stanford University, April 1993.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Int'l Symp. on Computer Architecture*, pages 15–26, May 1990.

[13] P. B. Gibbons and M. Merritt. Specifying nonblocking shared memories. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 306–315, 1992.

[14] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. *Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[15] A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations of Java memory behavior. Technical Report CS0922, Computer Science Department, Technion, November 1997.

[16] J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

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

[18] L. Higham and J. Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. of the 12th Int'l Symp. on Distributed Computing*, pages 201–215, September 1998.

[19] L. Higham, J. Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, October 1997.

[20] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, Department of Computer Science, The University of Calgary, January 1998.

[21] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part II: Process coordination problems. Technical Report 98/613/04, Department of Computer Science, The University of Calgary, January 1998.

[22] L. Higham and N. Verwaal. Processor consistency. In Preparation.

[23] J. James and A. Singh. The impact of hardware models on shared memory consistency conditions. *Lecture Notes in Computer Science*, 1119:719–734, 1996.

[24] P. Kohli, G. Neiger, and M. Ahamad. A characterization of scalable shared memories. In *Proc. of the 1993 Int'l Conf. on Parallel Processing*, August 1993.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.

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

[27] L. Lamport. On interprocess communication (parts I and II). *Distributed Computing*, 1(2):77–85 and 86–101, 1986.

[28] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

[29] D. Mosberger. Memory consistency models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

[30] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Proc. of the 7th ACM Symp. on Parallel Algorithms and Architectures*, July 1995.

[31] J. Protic, M. Tomasevic, and V. Milutinovic, editors. *Distributed Shared Memory Concepts and Systems.* IEEE CS Press, 1998.

[32] P. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report CSL-91-11, XEROX Corporation Palo Alto Research Center, December 1991.

[33] N. Verwaal. Ambiguous memory consistency models. M.Sc. Thesis, Department of Computer Science, The University of Calgary, 1998.

[34] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual version 9.* Prentice-Hall, 1994.