Defining and Comparing Memory Consistency Models*†

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

Abstract

Because multiprocessors implement sophisticated memory structures in order to enhance performance, processes can have inconsistent views of memory which may result in unexpected program outcomes. A memory consistency model is a set of guarantees that describes constraints on the outcome of sequences of interleaved and simultaneous operations in a multiprocessor. In this paper, we present a unifying framework to describe, understand, and compare memory consistency models. The framework is used to redefine and compare several widely used consistency models.

Key Words: memory consistency models,TSO,PSO.

1 Introduction

In order to enhance performance, multiprocessors tend to implement sophisticated memory structures. These memories may replicate data through constructs such as caches and write buffers. Furthermore, the time required to access a data object may vary between processes and between objects. 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 machines that are very complex to understand and program. It is thus essential to provide multiprocessor programmers with a precise description of the memory model of the underlying machine.

Several memory consistency models have been described in the literature. These descriptions arise from a wide variety of sources including architecture, system, and database designers, application programmers, and theoreticians. These descriptions use different types and degrees of formalism and hence are difficult to compare. Others are informal and sometimes ambiguous. There is no single unified formalization that describes the memory models addressed in the literature or provided by several existing machines.

In this paper, we present a unifying framework to describe, understand, and reason about memory consistency models. Precision is essential for providing an unambiguous view of the logical behaviour of a memory system. Unification provides a common basis by which memory consistency models can be compared.

The framework is defined in Section 2. Sections 3 and 4 use the framework to redefine and compare several widely used memory consistency models. In Section 3, we look at linearizability [10], sequential consistency [14], coherence [9], pipelined-RAM [15], and processor consistency [9]. Section 4 addresses models that make distinctions between kinds of accesses. These include SPARC's total store order and partial store order machines [18], and weak consistency [2]. Section 5 provides a summary and directions for further research.

2 The Model

Our goal is to have a framework that applies to any multiprocessor system of processes whether message-passing or shared-memory or any hybrid of the two. For any distributed system, this framework should be capable of describing the exact behaviour of the memory of that system. As shown in figure 1, we wish to model a multiprocessor system as a collection of processes operating on a collection of shared data objects.

Informally, the program of each process issues a stream of invocations to a collection of abstract objects and receives a collection of responses that, from the process's point of view, are interleaved with its stream of invocations. We do not specify how the objects are implemented, how the communication pro-

*©1997 ISCA. Published in the Proceedings of PDCS'97, October 1997 in New Orleans, USA.

†This research was supported in part by the Natural Sciences and Engineering Research Council of Canada in the form of trust funding and a post-graduate scholarship.

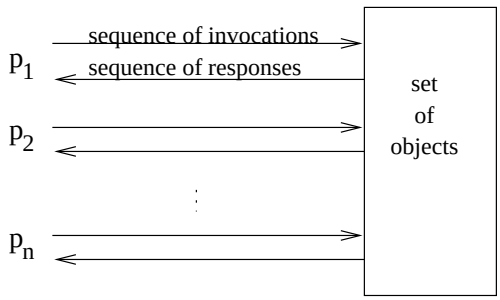

Figure 1: *A multiprocessor system*

ceeds or how the invocations are serviced. Instead we precisely define the constraints that the memory system imposes on the invocations and responses observed by each process. This is achieved by formalizing objects, processes, executions and the constraints on these executions.

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 *define* a data object to be the set of all sequence of allowable operations together with their results, 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 *(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 $(R, x, \lambda, (v_i))$[1], that returns a value v_i of register x, or a write action, denoted $(W, x, (v_i), \lambda)$, that assigns a value v_i 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 (op, obj, in, out) can be decomposed into two *matching* components, namely (op, obj, in) and (op, obj, out). The component (op, obj, in) is called the *action-invocation* and (op, obj, out) is the matching *action-response*. Given an action o = (op, obj, in, out), the action-invocation components of o are denoted invoc(o), and the action-response components are denoted resp(o). An *event* is either an action-invocation or an action-response. Let $(e_1, e_2, ...)$ be a sequence of events. Then event e_j *follows* event e_i if and only if $i < j$ and event e_j *immediately follows* event e_i if and only if $i = j - 1$.

A *process* is a sequence of action-invocations.

A *(multiprocess) system*, (P, J), is a collection P of processes and a collection J of objects, such that for every process $p \in P$ the action-invocations of p are applied to objects in J.

A *process execution* is a (possibly infinite) sequence of events, such that each response event follows its matching invocation event. A *(multiprocess) system execution* for a system (P, J), is a collection of process executions, one for each $p \in P$.

Note that an operation applied to an object may not produce an explicit output. (For example, an atomic write of a register by a process may not be followed by an acknowledgment of that write and thus the output component of the write action is empty.) We assume however, that in a process execution, each invocation event is followed by a matching response event by augmenting the stream of response events with implicit acknowledgement events for each invocation that does not have an explicit response. We place these implicit responses immediately following the matching invocation, thus reflecting the semantics of the process — that the invocation is assumed to be immediately executed. A process execution is *blocking* if each response event immediately follows its matching invocation event.

A system execution gives rise to a set of actions — namely the set of actions that result from combining each action-invocation with its matching action-response.

A *timed-action* on an object is a 6-tuple (op, obj, in, input-time, out, output-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 input-time (respectively, output-time) of the timed-action.

Let (P, J) be a multiprocess system, and O be the set of actions that result from an execution of this system. $O|p$ denotes the set of actions $o \in O$ such that invoc(o) is in the sequence of action-invocations that defines p in P. $O|x$ is the set of actions that are applied to object x in J.

We define two partial orders 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. The partial order $(O, \xrightarrow{prog})$ is called the *program order*. 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)) >

[1] λ denotes the empty sequence.

time(resp(o_1)). The partial order $(O, \xrightarrow{time})$ is called the *time order*. Observe that for each process $p \in P$, the program order is a total order on $O|p$.

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 be that subset of O consisting of those actions in O that update a shared object, and O_r be that subset consisting of the actions that only inspect a shared object. There are also some consistency models that provide other synchronization operations. Their corresponding actions are defined when needed.

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

In the following sections, we define various consistency conditions by stating restrictions on executions. An execution E meets some consistency condition C if the execution meets all the conditions of C. A system provides memory consistency C if every execution that can arise from the system meets the consistency condition C [2].

3 Memory Consistency Models

The literature describes many consistency conditions. 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 often are incomparable. We have chosen a selection here to illustrate the appropriateness and flexibility of our framework.

In the following example executions $w(x)v$ denotes a write of value v to variable x. Similarly, $r(x)v$ denotes a read from x returning v. We write the process identifier as a prefix for its own execution. Sometimes, we write $w_p(x)v$ or $r_p(x)v$ to emphasize that these operations are performed by process p.

3.1 Linearizability

Linearizability, the strongest correctness condition in the literature, was defined by Herlihy and Wing [10]. Mosberger calls linearizability *dynamic atomic consistency* [16]. An execution is linearizable if there is an assignment of each action to one distinct point after the action invocation, and before the action response on the time line such that the resulting sequential view of this execution is valid for each object.

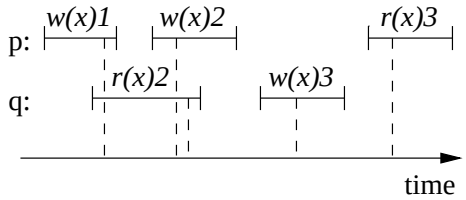

Figure 2: *Linearizable sequence of actions*

The execution in figure 2 shows an execution of the system with $P = \{p, q\}$ and $J = \{x\}$ that is linearizable. Notice that because the time interval of the $r(x)2$ 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) in figure 2 is valid for object x (assuming $x \neq 2$ initially).

Definition 3.1 *Let O be the set of actions that results from an execution E of the multiprocess system (P, J). Then E is* linearizable *if there is a linearization $(O, <_L)$ satisfying:*

1. $(O, \xrightarrow{prog}) \subseteq (O, <_L)$, *and*
2. $(O, \xrightarrow{time}) \subseteq (O, <_L)$.

Herlihy and Wing require that each process's execution is blocking, definition 3.1 extends to non-blocking executions while agreeing with the definition of Herlihy and Wing when the execution is blocking.

3.2 Sequential Consistency

Sequential Consistency (henceforth abbreviated SC), defined by Lamport [14], is the most widely used memory consistency model. According to Lamport:

"A multi process is said to be *sequentially consistent* if the result of any execution is the same as if the actions of all the processes were executed in some sequential order, and the actions of each individual process appear in this sequence in the order specified by its program."

SC is a weaker model of correctness than linearizability because the real time ordering of events does not have to be maintained. However, it must be possible to list all actions by all processes in the system in one linear order (a linearization) that agrees with

[2] The definitions of the various memory consistency models are given assuming that the multiprocess system terminates. They can be extended so that they apply to long-lived systems by using ideas similar to those of Herlihy and Wing [10, 17].

program order. Several other papers [4, 10, 9, 16] describe the same consistency condition possibly under a different name or using different definitions.

Definition 3.2 *Let O be the set of actions that results from the execution E of the multiprocess system (P, J). Then E is* sequentially consistent *if there is a linearization $(O, <_L)$ such that $(O, \stackrel{prog}{\longrightarrow}) \subseteq (O, <_L)$.*

Dubois, Scheurich and Briggs define *strong ordering* as a sufficient condition for SC [5] and Goodman states that "A system that adheres to this level of consistency is said to be a strongly ordered system" [9]. However, Adve and Hill show that strong ordering and SC are similar, but are not equivalent [2].

3.3 Coherence

Coherence, also called cache consistency, is among the weakest consistency conditions. Goodman states that coherence "only guarantees that accesses to a given memory location are strongly ordered" [9]. Similarly, Mosberger indicates that "Coherence only requires that accesses are SC on a *per-location* basis" [16]. These descriptions imply that an execution is coherent if, for each location, all read and write actions to that location can be ordered such that the resulting sequence is valid and that program order is maintained.

Definition 3.3 *Let O be the set of actions resulting from an execution E of the multiprocess system (P, J). Then E is* coherent *if for each object $x \in J$ there is some linearization $(O|x, <_{L_x})$ satisfying $(O|x, \stackrel{prog}{\longrightarrow}) \subseteq (O|x, <_{L_x})$.*

Execution 1 gives an example of an execution that 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 linearization of all these actions that maintains the program order.

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

Often coherence is described as a system where all processes view all the write actions to the same location in the same order [1, 8, 13] , or "all writes to the same location are serialized in some order and are performed in that order with respect to any processor" [8]. With the assumption that program order is maintained, this alternate definition is equivalent to definition 3.3 [17].

3.4 Pipelined-RAM

Lipton and Sandberg defined the Pipelined Random Access Machine (P-RAM) model of memory consistency [15]. Whereas coherence orders actions from the point of view of each object, P-RAM orders all actions from the process's point of view. Each process only "sees" all of its own actions and other process's writes. In P-RAM there must be, for each process, a linear ordering of its actions and all others' write actions that maintains program order and is valid. This does not mean that all processes will order all actions in the same way since the interleaving of the actions by different processes can be perceived differently by each process. P-RAM was introduced to capture a model where each process has a local copy of the memory. The propagation of writes to the processes could cause writes to arrive in different orders [16].

The following definition is based on that of Ahamed et al. [4] but reformulated using our framework:

Definition 3.4 *Let O be the set of actions that results from an execution E of the multiprocess system (P, J). The execution E is* P-RAM *if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying $(O|p \cup O_w, \stackrel{prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$.*

Execution 1 is P-RAM as well as coherent. Execution 2 is P-RAM but not coherent.

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

Let $<_p = w(x)0\ w(x)1\ r(x)1$ and $<_q = w(x)1\ w(x)0\ r(x)0$. Then $<_p$ (respectively, $<_q$) is a linearization of the actions by p (respectively, q) together with all the write actions, which maintains program order. Hence this execution is P-RAM. Since it is not possible to construct a linearization of all the actions to location x that maintains program order, it is not coherent.

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

Execution 3 is coherent but not P-RAM. 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 execution is coherent. It is, however, not possible to construct a linearization for the actions by q together with the writes by p that extends program order, and hence the execution is not P-RAM. These two examples show that P-RAM and coherence are incomparable consistency conditions.

3.5 Processor Consistency

The term Processor Consistency (PC) was first used by Goodman [9] to capture a consistency condition that is stronger than coherence but weaker than SC. Many others [4, 8, 13, 16, 7] have used the same term to define memory consistency models that have in common Goodman's original intentions, but that differ in subtle ways. We are preparing a separate work to reveal the relations and differences between these different versions of PC [11]. In this paper, we limit our discussion to the original definition by Goodman (denoted PCG). All PC definitions guarantee at least coherence, but are weaker than SC. In addition to coherence, Goodman[3] required [9]:

" 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. Again, the following definition is similar to that of Ahamed et al. [4].

Definition 3.5 *Let O be the set of actions that result from an execution E of a multiprocess system (P, J). Then E is* Processor Consistent according to Goodman *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. $\forall q \in P \ (O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$

By comparing this definition with definition 3.3 it is easily confirmed that PCG implies coherence [17].

Execution 4 $\begin{cases} p : w(x)1 \ w(y)1 \ r(y)0 \\ q : w(y)0 \ w(x)0 \ r(x)1 \end{cases}$

Consider execution 4. The following linearizations $<_{L_p} = w_p(x)1 \ w_p(y)1 \ w_q(y)0 \ r_p(y)0 \ w_q(x)0$ and $<_{L_q} = w_q(y)0 \ w_q(x)0 \ w_p(x)1 \ r_q(x)1 \ w_p(y)1$ show that execution 4 is P-RAM. However, it is not possible to build linearizations for p and q such that they agree on the ordering of write actions to the same location; therefore, execution 4 is not PCG. Furthermore, the first condition of PCG is exactly the definition of P-RAM. Therefore PCG is strictly stronger than P-RAM.

Although PCG implies both P-RAM and coherence, an execution that is both coherent and P-RAM is not necessarily PCG as illustrated by execution 5.

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

For each location there is a linearization $(O|x, <_{L_x})$ that maintains the program order. In execution 5, let $<_{L_x} = w(x)1 \ r(x)1 \ w(x)0 \ r(x)0$ and let $<_{L_y} = w(y)0 \ r(y)0$. These are linearizations for each object that maintain program order. Now consider the sequences: $<_{L_p} = w(x)1 \ w(x)0 \ w(y)0$, $<_{L_q} = w(x)0 \ w(y)0 \ r(y)0 \ w(x)1$, and $<_{L_r} = w(x)1 \ r(x)1 \ w(x)0 \ r(x)0 \ w(y)0$. These linearizations establish that the execution is P-RAM. However, notice that it is necessary to change the order in which the writes to location x were perceived by q and r to get a linearization for each. In order for an execution to be PCG, the writes to the same object must be perceived by each process in the same order as given by the linearization for that object. Therefore, execution 5 is not PCG.

Finally, execution 1 is PCG, establishing that PCG is strictly weaker than SC.

4 Hybrid Models

Hybrid models distinguish operations according to their function. Operations are classified as special or ordinary. Special operations may be divided into synchronization or non-synchronization. Synchronization operations may be further classified into acquires and releases. Further classification has been proposed [6]. These models aim at utilizing system optimizations while still appearing sequentially consistent to the programmer. This is achieved by requiring the programmer to (sometimes conservatively) label program operations according to their classifications. Gharachorloo et al.[6] used this idea in what they called the programmer-centric approach. In this paper, only the memory models themselves are explored without considering the programming style.

The hybrid models described in this section are weak consistency, and SPARC's total store ordering and partial store ordering. More hybrid models have been proposed in the literature. These include DRF0 [2], DRF1 [3], release consistency [8, 7], PL_{SC} [8], and PL_{PC} [6]. These models can be similarly described using our framework.

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

The formal definitions of the 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 hidden by more informal definitions.

4.1 Weak Consistency

Dubois, Scheurich and Briggs were the first to propose weak consistency (WC) [5] (Adve and Hill call it weak ordering [2]). A WC system distinguishes between ordinary and synchronization operations, and guarantees minimum constraints for each class. The synchronization operations must satisfy sequential consistency, and all processes must view an ordinary action before (respectively after) a synchronization action if they are ordered so in program order. Gharachorloo et al. [8] define WC as follows:

1. before an ordinary LOAD or STORE access is allowed to perform with respect to any other process, all previous *synchronization* accesses must be performed, and

2. before a *synchronization* access is allowed to perform with respect to any process, all previous ordinary LOAD and STORE accesses must be performed, and

3. *synchronization* accesses are sequentially consistent with respect to one another.

In addition, Gharachorloo implies that WC maintains coherence and that each process views its own operations in program order.

The definition of WC using our framework follows. Let O_{synch} denote the set of synchronization actions. Let E be a coherent execution of system (P, J) with resulting set of actions O, and $\forall x \in J$ let $(O|x, <_{L_x})$ be the linearization guaranteed by coherence. Define $o_1 \overset{weak<_{L_x}}{\longrightarrow} o_2$ by $o_1 <_{L_x} o_2$ and one of $\{o_1, o_2\}$ is a read action and one is a write action. Define $o_1 \overset{weak-prog}{\longrightarrow} o_2$ by $o_1 \overset{prog}{\longrightarrow} o_2$ and at least one of $\{o_1, o_2\}$ is a synchronization action. Now define $\forall o_1, o_2 \in O$ $o_1 \overset{wc}{\longmapsto} o_2$ iff either

1. $o_1 \overset{weak-prog}{\longrightarrow} o_2$ or

2. $\exists o' \in O$ such that $o_1 \overset{weak-prog}{\longrightarrow} o' \overset{weak<_{L_x}}{\longrightarrow} o_2$ or

3. $\exists o'$ such that $o_1 \overset{wc}{\longmapsto} o' \overset{wc}{\longmapsto} o_2$.

Definition 4.1 *Let O be the set of actions that result from an execution E of a multiprocess system (P, J). Then E is* weakly consistent *if for each process $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ satisfying*

1. $(O|p, \overset{prog}{\longrightarrow}) = (O|p, <_{L_p})$, *and*

2. $\forall q \in P$ $(O|x \cap O_w, <_{L_p}) = (O|x \cap O_w, <_{L_q})$, *and*

3. $(O|p \cup O_w, \overset{wc}{\longmapsto}) \subseteq (O|p \cup O_w, <_{L_p})$, *and*

4. $\forall q \in P$ $(O_w \cap O_{synch}, <_{L_p}) = (O_w \cap O_{synch}, <_{L_q})$

4.2 SPARC v8

The SPARC v8 [18] architecture implements, selectively, two memory models: Total Store Ordering (TSO) and Partial Store Ordering (PSO). SPARC multiprocessors utilize store buffers; one is associated with each processor in the system. Each buffer operates in parallel with the processor, which does not wait (block) for a store to be committed to main memory. Instead, the processor inserts the store into the buffer (we call this a *buffer store*). The store buffer subsequently commits pending stores to main memory (we call this a *memory store*). The order in which buffer stores are committed to main memory differentiates between TSO and PSO. When the buffer behaves as a First In First Out (FIFO) queue, the model is called TSO. The model is PSO, when it is only guaranteed that stores performed to the same location are sent to main memory in FIFO order. In both models, when a load to location, say l, is executed, the processor's own store buffer is consulted to see if the latest store to l is still pending. In this case, the load returns the value to be written by that pending store to l. If the buffer contains no stores to l, main memory is accessed.

In SPARC, main memory accepts one memory action at a time. A bus (or a nondeterministic switch) serves one processor memory request at a time. This ensures that memory stores of the store buffers are executed sequentially[4].

TSO and PSO both include the special operation *swap atomic* [18], which is both a load and a store. Since it has write[5] semantics, it is thrown into the buffer as any other store, and since it has read semantics, every other following load has to wait until the

[4] Overlapping loads or buffer stores by two processors can be assumed to occur in an arbitrary sequential order since the outcome of the computation would be unaffected by such sequentialization.

[5] The architecture of the SPARC v8 is described in terms of *loads* and *stores*. We specify constraints on executions in terms of *reads* and *writes*.

swap completes (commits to main memory and returns the loaded value). Thus, a swap atomic in TSO blocks the processor and flushes the store buffer.

PSO also includes an additional special action, a *store barrier*, which is also inserted into the store buffer [18]. This fence instruction is used to impose additional constraints on the ordering of stores. No store issued after (in program order) a store barrier is allowed to complete before a store that is issued before (in program order) the barrier. Let $(O_w, \overset{writes}{\longrightarrow})$ be a total order on all the writes of an execution ordered in the same order as the corresponding memory stores. Each processor "sees" writes by other processors. The writes of a processor p, however, are visible to p (as buffer stores) before they are visible to other processors. It is also possible that a write by some other processor is invisible to p. This happens if there is a location l and processor q, such that the following sequence takes place in this order: p buffer stores a value v to location l, q memory stores v' to l, then, p memory stores v to l. In this case, the write by q to l is never visible to p. These observations provide the intuition for definition 4.2. In the following, $(A \uplus B)$ denotes the disjoint union of sets 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 the set of swap atomic actions. Note that, $O_w \cap O_r = O_a$ where O_r denotes the set of read actions.

Definition 4.2 *Let O be the set of actions that results from an execution E of the multiprocess system (P, J). Then E is TSO if there exists a total order $(O_w, \overset{writes}{\longrightarrow})$ such that $(O_w, \overset{prog}{\longrightarrow}) \subseteq (O_w, \overset{writes}{\longrightarrow})$ and $\forall p \in P$ there is a total order $(O|p \uplus O_w, \overset{merge_p}{\longrightarrow})$, satisfying:*

1. $(O|p, \overset{prog}{\longrightarrow}) = (O|p, \overset{merge_p}{\longrightarrow})$, *and*
2. $(O_w, \overset{writes}{\longrightarrow}) = (O_w, \overset{merge_p}{\longrightarrow})$, *and*
3. *if $w \in (O|p \cap O_w)$ (i.e. necessarily, a write by p) 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*
   $O_{invisible_p} = \{w \mid w$ *is a write by* $q \neq p$ *to* $x \wedge \exists w' \in O|x \cap O|p \cap O_w \wedge w'_{O|p} \overset{merge_p}{\longrightarrow} w \overset{merge_p}{\longrightarrow} w'_{O_w}\}$
   $O_{memwrites_p} = \{w_{O_w} \mid w \in O|p \cap O_w\}$, *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 $w_{O_w} \overset{merge_p}{\longrightarrow} a$.*

PSO provides fewer guarantees than TSO; specifically, the total order over all writes imposed by TSO is true for PSO only if these writes are to the same location. Therefore, a swap atomic flushes the buffer in PSO only if all pending stores (including the swaps) are to the same location. For this reason, swap atomics need no exclusive treatment in our definition (recall that $O_a \subseteq O_w$). Let O_{sb} denote the set of store barrier actions.

Definition 4.3 *Let O be the set of actions that results 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 $w_{O_w} \overset{merge_p}{\longrightarrow} u_{O_w}$.*

Any TSO execution is also PSO. We wish to compare TSO and PSO with the other models using only ordinary actions. Thus, we assume $O_a = \emptyset$ in the following discussion.

For a write $w(x)v \in O|p \cap O_w$, let $\widehat{w}(x)v$ denote the copy in $O|p$, and let $\overline{w}(x)v$ denote the copy in O_w.

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

Let $(O_w, \overset{writes}{\longrightarrow})$ be given by the ordering: $w_q(y)1$, $w_q(y)2$, $w_q(x)6$, $w_q(y)3$, $w_p(x)0$. Let $(O|p \uplus O_w, \overset{merge_p}{\longrightarrow})$ be given by the ordering: $\widehat{w}_p(x)0$, $w_q(y)1$, $r_p(y)1$, $w_q(y)2$, $r_p(y)2$, $w_q(x)6$, $w_q(y)3$, $r_p(y)3$, $r_p(x)0$, $\overline{w}_p(x)0$ and $(O|q \uplus O_w, \overset{merge_q}{\longrightarrow})$ be given by the ordering: $\widehat{w}_q(y)1$, $\overline{w}_q(y)1$, $\widehat{w}_q(y)2$, $\overline{w}_q(y)2$, $\widehat{w}_q(x)6$, $\overline{w}_q(x)6$, $\widehat{w}_q(y)3$, $\overline{w}_q(y)3$, $w_p(x)0$.

It is easily checked that conditions 1, 2, and 3 of definition 4.2 hold. Notice that $O_{invisible_p} = \{w_q(x)6\}$ and $O_{memwrites_p} = \{\overline{w}_p(x)0\}$. Omitting the actions in these two sets from the ordering $(O|p \uplus O_w, \overset{merge_p}{\longrightarrow})$ given above results in a linearization. Also, $O_{invisible_q} = \emptyset$ and $(O|q \uplus O_w \backslash O_{memwrites_q}, \overset{merge_q}{\longrightarrow})$ is a (trivial) linearization. Therefore, condition 4 of definition 4.2 also holds, and Execution 6 is TSO.

However it is not P-RAM. Notice that to maintain validity and program order, we need: $w(y)1 <_{L_p} r(y)1 <_{L_p} w(y)2 <_{L_p} r(y)2 <_{L_p} w(y)3 <_{L_p} r(y)3$. Therefore, to find a linearization $<_{L_p}$ preserving $\overset{prog}{\longrightarrow}$ over $(O|p \cup O_w)$, we have $w(x)0 \overset{prog}{\longrightarrow} r(y)1 <_{L_p} w(y)2 \overset{prog}{\longrightarrow} w(x)6 \overset{prog}{\longrightarrow} w(y)3 <_{L_p} r(y)3 \overset{prog}{\longrightarrow} r(x)0$. This resolves to $w(x)0 <_{L_p} w(x)6 <_{L_p} r(x)0$ which is invalid. Since TSO (PSO) is not P-RAM, it is neither PCG nor SC.

Any TSO (PSO) execution is coherent. The proofs are not trivial [12].

5 Conclusion and Further Work

We introduced a formal framework for describing memory consistency models which applies to any multiprocessor system. The framework unifies the language by which different memory consistency models can be described and compared. To illustrate the usefulness of our framework, we have redefined several widely known memory consistency models. These are linearizability, sequential consistency, coherence, processor consistency, pipelined-RAM, weak ordering, total store ordering, and partial store ordering. The formalism in our framework was exploited to compare these different models.

Future research will be directed towards further exploitation of the formalism provided here in order to map one memory consistency model to another. Such a mapping will allow programmers to design and reason about programs in a sequential consistent model while eventually running their programs on machines with weaker memory systems.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] L. Higham and N. Verwaal. Processor consistency. In preparation.

[12] J. Kawash and L. Higham. Memory consistency models of bus-based multiprocessors. Technical Report 96/594/14, Department of Computer Science, The University of Calgary, October 1996.

[13] 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.

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

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

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

[17] N. Verwaal. M.sc. thesis draft. verwaal@cpsc.ucalgary.ca, The University of Calgary.

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