

# Memory Consistency and Process Coordination for SPARC Multiprocessors

Lisa Higham\* and Jalal Kawash\*\*

Department of Computer Science, The University of Calgary, Canada, T2N 1N4  
{higham|kawash}@cpsc.ucalgary.ca

*In the proceedings of the 7th International Conference on High Performance Computing (HIPC 2000), India, December 2000, Lecture Notes in Computer Science #1970, pages 355 - 366, Springer.*

**Abstract.** Simple and unified non-operational specifications of the three memory consistency models Total Store Ordering (TSO), Partial Store Ordering (PSO), and Relaxed Memory Order (RMO) of SPARC multiprocessors are presented and proved correct. The specifications are intuitive partial order constraints on possible computations and are derived from natural successive weakening of Lamport's Sequential Consistency. The formalisms are then used to determine the capabilities of each model to support solutions to critical section coordination and both set and queue variants of producer/consumer coordination without resorting to expensive synchronization primitives. Our results show that none of RMO, PSO nor TSO is capable of supporting a read/write solution to the critical section problem, but each can support such a solution to some variants of the producer/consumer problem. These results contrast with the two previous attempts to specify these machines, one of which would incorrectly imply a read/write solution to the critical section problem for TSO, and the other of which is too complicated to be useful to programmers. Our general framework for defining and proving the correctness of the memory consistency models was key in uncovering the previous error and in achieving our simplification, and hence may be of independent interest.

## 1 Introduction

Sun Microsystems introduced the three memory consistency models Total Store Ordering (TSO), Partial Store Ordering (PSO), and Relaxed Memory Order (RMO). TSO and PSO were introduced in the version 8 architecture[16] and retained in version 9. RMO was introduced in the version 9 architecture[17]. These models are specified axiomatically — some partial orders are defined, relationships between them are specified, and an axiom given that determines the values returned by reads. These specifications are complicated and not intuitive. Hence it is challenging for a programmer

---

\* Supported in part by the Natural Sciences and Engineering Research Council of Canada grant OGP0041900.

\*\* Supported in part by a Natural Sciences and Engineering Research Council of Canada doctoral scholarship and an Izaak Walton Killam Memorial scholarship.

to be certain of the possible program outcomes and to determine correct and efficient solutions to basic process coordination problems.

This dilemma is typical of current and proposed multiprocessor machines. Weakening the memory consistency model of a multiprocess system improves its performance and scalability. However, these models sacrifice programmability because they create complex behaviors of shared memory. Without the use of expensive, built-in synchronization, these models exhibit poor capabilities to support solutions for fundamental process coordination problems [9]. This leads programmers to aggressively use these forms of synchronization, incurring additional performance burdens on the system.

Since use of synchronization primitives deteriorates performance, we are motivated to study the limitations and capabilities of weak memory consistency models without the use of these primitives. If the use of explicit synchronization is avoidable, then efficient libraries for certain classes of applications can be built. This would ease the job of distributed application programmers and make their applications more efficient.

In this paper, we derive a mathematical description of the behavior of the three SPARC variants in terms of partial order constraints on possible computations. Our specifications are surprisingly simple; each is a natural weakening of Sequential Consistency (SC) [12]. Because they are short and precise, we believe they give programmers a much improved tool for reasoning about the outcomes of their multiprocess programs. They also make it easier to construct verification tools that depend on automated reasoning. Finally, the simplicity of the descriptions facilitates the comparison of the SPARC models to each other and to several proposed consistency models including Processor Consistency[6], Causal Consistency[1], and Java Consistency[5, 7].

Our definitions for TSO and PSO are proven to exactly capture the machine description of the version 8 architecture (and hence of version 9 since they are known to be equivalent). Our RMO definition is proven to exactly capture the version 9 specification. To achieve our proofs we establish a framework for describing operational and non-operational models and a technique for proving their equivalence. Because this framework is not restricted to any particular machine, it may be of independent interest. We have used it to prove correctness of other models [9].

We next use our partial order specifications to study the capabilities of the three SPARC memory consistency models TSO, PSO, and RMO to support solutions to fundamental process coordination problems without resorting to expensive synchronization primitives. The process coordination problems studied in this paper are critical section coordination and producer/consumer coordination. We distinguish two variants of producer/consumer coordination whose solution requirements differ: the set and queue variants. Our results show that the TSO (and hence PSO and RMO) model is incapable of supporting a read/write solution to the critical section problem, but even RMO (and hence PSO and TSO) can support such solutions to some variants of the producer/consumer problem.

Capturing the semantics of the three SPARC memory consistency models simply and precisely has proven to be surprisingly tricky. One earlier attempt to define TSO [11] resulted in a definition that is much stronger than what this architecture really provides and leads to erroneous conclusions about the coordination capabilities of TSO. In fact, we show that any program (with only read/write operations) that is correct for SC

can be compiled into an equivalent program (with only read/write operations) that is correct for this erroneous version of Total Store Ordering [10]. To the contrary, this paper proves that read/write operations are insufficient to solve certain coordination problems for TSO. Another earlier definition for Total Store Ordering is one of our own that is also based on partial orders. Though it is proved equivalent to TSO [10], it is much more complicated than TSO as defined here and it is not completely non-operational. The original operational specifications of TSO and PSO [16] are also complex and are not particularly useful for studying the questions addressed in this paper. The subsequent update to version 9 definitions and the extension to RMO is renowned for the complexity of its many partial orders and especially of the value axiom. Park and Dill also study the Relaxed Memory Order [14]. The goal of their work is a verification tool for RMO constructed from the Mur $\phi$  language, rather than a reformulation of the specifications. Their definition of RMO is taken unchanged from the SPARC Architecture Manual v9 .

Section 2 contains one of our main contributions — simple, unified, non-operational models of the SPARC TSO, PSO and RMO memory consistencies. The SPARC version 8 machine is overviewed in Section 3 in order to provide the basis of the proofs that our non-operational models for TSO and PSO are correct. Section 4 sketches this proof for TSO and overviews the corresponding proofs for PSO and RMO. Section 5 briefly examines the error with the earlier attempt at a simple partial order to capture the semantics of TSO. Section 6 contains our other main results. It establishes the possibilities and impossibilities for process coordination problems in SPARC TSO, PSO, and RMO machines. The SPARC models are compared with others in Section 7.

This presentation addresses only read and write operations to variables. SPARC machines support many other operations that affect shared memory such as swap-atomic and barrier (version 8) and the family of membar operations (version 9). It is straightforward to extend our models and our proofs to include these operations. Several proofs are omitted here and other only sketched. The full version of this paper [8] contains the complete models and all proofs.

## 2 The SPARC Memory Consistency Models

This section defines three memory consistency models, called TSO, PSO, and RMO based on progressive weakening of SC. The definitions refer only to the ordering of operations; they are very simple and natural; they do not depend on a machine description. We will see in Section 4, however, that these definitions do indeed capture exactly the behavior of the corresponding SPARC architectures.

We first summarize the way that we formalize any memory consistency model. A multiprocess system can be modeled as a collection of processes operating on a collection of shared data objects. The only shared data objects considered here are variables supporting read and write operations. The full version of this paper [8] provides the extension to other SPARC operations. The notation  $r(x)v$  and  $w(x)u$  denotes, respectively, a read operation of variable  $x$  returning  $v$  and a write operation to  $x$  of value  $u$ . The *invocation of operation  $o$* , is just the operation without its output value determined and is denoted  $in(o)$ . A read  $r(x)v$  and a read invocation with  $v$  as yet undetermined, are

distinguished by writing  $in(r(x)v) = r(x)v?$ . Since a write invocation has no response value,  $in(w(x)u) = w(x)u$ .

It suffices to model a *process* as a sequence of read and write invocations, and a *multiprocess system* as a collection of processes together with the shared variables. Henceforth, we denote a multiprocess system by the pair  $(P, J)$  where  $P$  is a set of processes and  $J$  is a set of variables. A *process computation* is the sequence of operations obtained from the process by augmenting each read operation invocation with its matching response. A *system computation* is a collection of process computations, one for each process. Let  $O$  be all the (read and write) operations in a computation of a system  $(P, J)$ . Then,  $O|p$  denotes all the operations that are in the process computation of process  $p \in P$ ;  $O|x$  denotes all the operations that are applied to variable  $x \in J$ ;  $O|w$  denotes all the write operations; and  $O|r$  denotes all the read operations. These notations are also combined to select the combined restriction of operations. For example,  $O|w|x|p$  is the set of all write operations by process  $p$  to variable  $x$ .

A sequence of read and write operations to variable  $x$  is *valid* if and only if each read operation in the sequence returns the value written by the most recently preceding write operation. Given any collection of read and write operations  $O$  on a set of variables  $J$ , a *linearization* of  $O$  is a (strict) linear order  $(O, \xrightarrow{L})$  such that for each variable  $x$  in  $J$ , the subsequence  $(O|x, \xrightarrow{L})$  of  $(O, \xrightarrow{L})$  is valid. For any relation  $R$ , the notation  $s_1 R s_2$  means  $(s_1, s_2) \in R$ . A read operation is *foreign* if the value returned by the read was written by a process different from the one invoking the read. Otherwise, it is called *domestic*.

Let  $O$  be a set of operations in a computation of a system  $(P, J)$ . Define the *program order*, denoted  $(O, \xrightarrow{prog})$ , by  $o_1 \xrightarrow{prog} o_2$  if and only if  $o_2$  follows  $o_1$  in the computation of  $p$ . Consider the following conditions on  $o_1$  and  $o_2$  where  $o_1 \xrightarrow{prog} o_2$ :

**same-variable:**  $o_1, o_2 \in O|x$ , for some  $x \in J$ .

**preceding-read:**  $o_1 \in O|r$  and  $o_1$  is foreign.

**following-write:**  $o_2 \in O|w$ .

The *RMO partial program order*, denoted  $(O, \xrightarrow{rmo})$ , is the transitive closure of the relation:  $(O, \xrightarrow{prog})$  intersect same-variable.

The *PSO partial program order*, denoted  $(O, \xrightarrow{pso})$ , is the transitive closure of the relation:  $(O, \xrightarrow{prog})$  intersect (either same-variable or preceding-read).

The *TSO partial program order*, denoted  $(O, \xrightarrow{tso})$ , is the transitive closure of the relation:  $(O, \xrightarrow{prog})$  intersect (either same-variable or preceding-read or following-write).

**Definition 1.** Let  $O$  be all the operations of a computation  $C$  of a multiprocess system  $(P, J)$ . Then

$C$  is TSO if there exists a linearization  $(O, \xrightarrow{L})$  such that  $(O, \xrightarrow{tso}) \subseteq (O, \xrightarrow{L})$ ,

$C$  is PSO if there exists a linearization  $(O, \xrightarrow{L})$  such that  $(O, \xrightarrow{pso}) \subseteq (O, \xrightarrow{L})$ ,

$C$  is RMO if there exists a linearization  $(O, \xrightarrow{L})$  such that  $(O, \xrightarrow{rmo}) \subseteq (O, \xrightarrow{L})$ .

Notice that RMO, when restricted to reads and writes, is just the well-known consistency model known as Coherence [3].

### 3 The SPARC v8 Multiprocessor Machines

In SPARC v8 (a two process version is in Figure 1) there is one write-buffer associated with each process in the system. The main memory is single ported with a non-deterministic switch providing one memory access at a time. When a process performs a write, it is sent to its write-buffer, which is responsible for committing the pending writes to main memory. The process, in the meantime, can continue executing. When a read is issued by a process, the associated write-buffer is checked for any pending writes to the same variable. If there is any such write, the value “to be written” by the last such write is returned. If there is no such pending write in the buffer, the read accesses main memory in the normal manner. The order in which the write-buffer commits the writes to main memory differentiates between two machine variants of the SUN SPARC v8. If the buffer is FIFO, the resulting machine is called Total Store Ordering and is denoted  $M_{TSO}$ . If it is FIFO only on a per-variable basis, the resulting machine is Partial Store Ordering, denoted  $M_{PSO}$ . SPARC v8 does not define a Relaxed Memory Order machine.

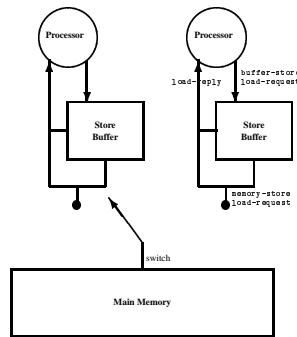


Fig. 1. A two-process SPARC architecture

To describe the precise operational behavior of the SPARC machine, we specify the sequence of events that is triggered by each operation invocation and the rules that constrain how these events interleave and instantiate variables. This description follows in a straightforward way from the description provided by SUN Microsystems [16]. Because of page limitations, we only provide the details for  $M_{TSO}$ .

The  $M_{TSO}$  is a machine that accepts read and write operation invocations, and for each executes a sequence of events.  $M_{TSO}$  is specified by the triple  $(\mathcal{E}, I, \mathcal{R})$  as follows.

$\mathcal{E}$ , the set of *event types*, contains load-request, load-reply, buffer-store, and memory-store. A `load-request`( $p, x$ ) is a request by process  $p$  to load the value stored in variable  $x$ . The event `load-reply`( $p, x, v$ ) returns a value,  $v$ , of  $x$  to  $p$ . The event `buffer-store`( $p, x, u$ ) is a store request by  $p$  to  $x$  of a value  $u$ ; the request is placed in the store-buffer associated with  $p$ . A `memory-store`( $p, x, u$ ) commits  $p$ 's request, `buffer-store`( $p, x, u$ ), by removing the request from the buffer and applying it to main memory.

$I$ , the *implementation function*, defines the sequence of two events that occurs for each operation invocation.

- A write in  $M_{TSO}$  is implemented by the ordered pair that represents placing the write in the buffer and later committing it to main memory.

$$I(w(x)u, p) = \langle \text{buffer-store}(p, x, u), \text{memory-store}(p, x, u) \rangle$$

- A read is implemented by an ordered pair that represent a request for a value followed by return of the requested value.

$$I(r(x)v?, p) = \langle \text{load-request}(p, x), \text{load-reply}(p, x, v?) \rangle$$

The events that implement an operation invocation *correspond* to that operation and two such events are *matching*.

$\mathcal{R}$ , the set of *machine rules*,  $\{\rho_i | 1 \leq i \leq 4\}$ , further restricts the machine behavior as follows. The implementation function,  $I$ , produces an ordered pair of two events for each read or write in  $(P, J)$ . The set  $E$  of all such events occur in some total order  $(E, \xrightarrow{\Xi})$  that respects the ordering of these pairs and the additional constraints  $\rho_1$ ,  $\rho_2$  and  $\rho_3$  and has variables instantiated according to  $\rho_4$ .  $(E, \xrightarrow{\Xi})$  is also denoted by  $\Xi$ . Recall that  $\xrightarrow{\text{prog}}$  denotes program order.

- $\rho_1$  (buffers are FIFO): Let  $e_1$  and  $e_2$  be buffer-store events by the same process. Also, let  $e'_1$  and  $e'_2$  be the matching memory-store events of  $e_1$  and  $e_2$ , respectively. If  $e_1 \xrightarrow{\Xi} e_2$  then  $e'_1 \xrightarrow{\Xi} e'_2$ .
- $\rho_2$  (loads are blocking): Let  $e$  represent a load-request and  $e'$  represent its matching load-reply event. If there is an event  $\hat{e}$  by the same process such that  $e \xrightarrow{\Xi} \hat{e} \xrightarrow{\Xi} e'$ , then  $\hat{e}$  is necessarily a memory-store event.
- $\rho_3$  (request order matches program order): Let  $I(\text{in}(o_1), p) = \langle e_1, e'_1 \rangle$  and  $I(\text{in}(o_2), p) = \langle e_2, e'_2 \rangle$ . If  $o_1 \xrightarrow{\text{prog}} o_2$  then  $e_1 \xrightarrow{\Xi} e_2$ .
- $\rho_4$  (variable instantiation): A  $\text{load-reply}(p, x, v?)$  event  $e$  instantiates  $v?$  as follows. If the most recent buffer-store that precedes  $e$  in  $\Xi|p|x$  is  $\text{buffer-store}(p, x, u)$  such that its matching  $\text{memory-store}(p, x, u)$  follows  $e$  in  $\Xi|x$ , then we have  $\text{load-reply}(p, x, v? \leftarrow u)$ . Otherwise,  $\text{load-reply}(p, x, v? \leftarrow u)$  where the most recent memory-store event that precedes  $e$  in  $\Xi|x$  is  $\text{memory-store}(q, x, u)$ .

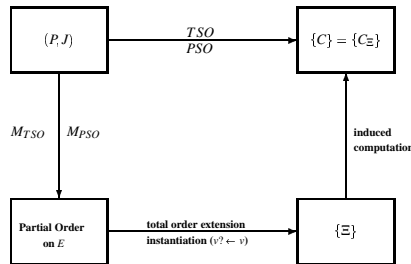
Call any such sequence of events  $(E, \xrightarrow{\Xi})$  that is generated by  $I$  acting on  $(P, J)$  and satisfying  $\mathcal{R}$ , an *execution* of  $(P, J)$  on  $M_{TSO}$ . An execution,  $\Xi$ , of the system  $(P, J)$  induces a computation  $C_\Xi$  of  $(P, J)$  — simply the one that attaches to each read invocation of the system the response value of the corresponding load-reply event.

## 4 Correctness of the SPARC Definitions

This section illustrates one of our proof techniques by sketching the proof that the machine  $M_{TSO}$  of Section 3 and the memory consistency model TSO of Section 2 are equivalent in the sense of the commuting diagram in Figure 2. Our goal is to prove that a computation of a system  $(P, J)$  could arise from an execution on  $M_{TSO}$  if and only if that computation satisfies TSO. This goal is achieved through two lemmas. Let  $(P, J)$  be a multiprocess system with read and write operation invocations. If a read  $r$  returns the value written by a write  $w$ ,  $r$  and  $w$  are said to be *causally-related*.

**Lemma 1.** Any computation that is induced by an execution of the system  $(P, J)$  on  $M_{TSO}$  is TSO.

**Proof Sketch:** Let  $O$  be all the operations of a computation  $C_{\Xi}$  that is induced by an execution  $\Xi$  of the system  $(P, J)$  on  $M_{TSO}$ . First, construct a sequence of operations  $X$  from  $\Xi$  as follows. For each `load-reply`( $-, x, v$ ) (respectively, `memory-store`( $-, x, u$ )) event in  $\Xi$  place a  $r(x)v$  (respectively,  $w(x)u$ ) operation in  $X$ , with read and write operations ordered in  $X$  as the corresponding load-replies and memory-stores are ordered in  $\Xi$ . For a read that completed at main memory, the corresponding load-reply instantiates its parameter variable according to a preceding memory event. So, this construction of  $X$  guarantees that such a read (domestic or foreign) follows its causally-related write. This does not hold for a read that corresponds to a load-reply that completed at the buffer level. Such a read precedes its causally-related write in  $X$ , violating both validity and  $\xrightarrow{TSO}$ . Because its causally-related write is (necessarily) applied to the same variable as the read, the program order of such a write followed by that read must be maintained in  $\xrightarrow{TSO}$  (same-variable condition). Now we adjust  $X$  by moving the domestic reads that violate validity as follows. Initially, mark every domestic read operation in  $X$  as *unvisited*. Iterate through  $X$  examining each unvisited domestic read operation  $o$  in turn. Let  $e$  be  $o$ 's corresponding load-reply event in  $\Xi$ . Let  $e'$  be the memory event in  $\Xi$  such that  $e$  returns the value written by  $e'$ . By construction of  $X$ ,  $e'$  has a corresponding write operation  $o'$  in  $X$ . If  $o'$  precedes  $o$  in  $X$ , then mark  $o$  as visited and continue with the first unvisited read in  $X$ . If  $o'$  follows  $o$  in  $X$ , then move  $o$  in  $X$  such that it immediately follows the latest of  $o'$  or the last moved read operation. Mark  $o$  as visited, and continue with the first unvisited read in  $X$ . Finally, define  $(O, \xrightarrow{L})$  to be this adjusted  $X$ . The proof proceeds by showing with careful case analysis that  $(O, \xrightarrow{L})$  is a linearization that extends  $(O, \xrightarrow{TSO})$ . ■



**Fig. 2.** Proving the correctness of TSO and PSO

**Lemma 2.** Any TSO computation of the system  $(P, J)$  is a computation induced by some execution of the system  $(P, J)$  on  $M_{TSO}$ .

**Proof Sketch:** Let  $C$  be a TSO computation of system  $(P, J)$ . We construct an execution  $\Xi$  of  $(P, J)$  on  $M_{TSO}$  that induces the computation  $C$ . Let  $O$  be all the operations resulting from  $C$ .

Given the linearization  $(O, \xrightarrow{L})$  guaranteed by Definition 1 (for TSO), first construct a sequence  $X$  of operations as follows. Initially  $X$  is empty. For each process  $p$  computation of  $C$ , maintain a pointer  $\downarrow_p$  that initially points at the first operation in  $p$ 's computation. If  $\downarrow_p$  points at  $o$ , then we say  $\downarrow_p = o$ . Similarly, maintain the pointer  $\downarrow_L$  to operations in  $(O, \xrightarrow{L})$ . Initially,  $\downarrow_L$  points at the first operation in  $(O, \xrightarrow{L})$ . When there are no more operations to consider in a sequence  $s$ , we say  $\downarrow_s = \perp$ . Also, advancing  $\downarrow_s$  means the pointer is incremented to point at the next operation in  $s$ . Initially, all operations are *unmarked*. Let  $\hat{o}$  denote the marked copy of operation  $o$ .

**Repeat until**  $\downarrow_L = \perp$ :

**If**  $\downarrow_L = o \in O|p$  for some  $p$ , and  $\downarrow_p = o$  **then**

Append  $o$  to  $X$ . Advance both  $\downarrow_L$  and  $\downarrow_p$ .

**Else-if**  $\downarrow_L = o \in O|p$  and  $o$  is unmarked but  $\downarrow_p = o' \neq o$  **then**

Append  $o'$  to  $X$ , and record  $o'$  as marked (denoted  $\hat{o}'$ ). Advance  $\downarrow_p$  only.

**Else** ( $\downarrow_L = o$  and  $o$  is marked)

**If**  $o$  is a write, **then** append  $\hat{o}$  to  $X$ .

Advance  $\downarrow_L$ .

In each iteration of the repeat loop, either  $\downarrow_L$  is advanced or an operation in  $(O, \xrightarrow{L})$  is marked, and no operation is marked more than once. Furthermore,  $\downarrow_L$  always advances over marked operations. Therefore this procedure terminates for finite computations. Notice that some writes are duplicated in  $X$  with the unmarked copy preceding the marked one. At the end (when  $\downarrow_L = \perp$ ), if there are write operations that have not been duplicated in  $X$ , then for each such write  $o$  insert a marked copy  $\hat{o}$  immediately after  $o$ .

Now, construct  $\Xi$  from  $X$  as follows. Iterating through  $X$ , consider each operation  $o$ . If  $o$  is a  $r(x)v$  by  $p$ , then append to  $\Xi$  `load-request` ( $p, x$ ) immediately followed by `load-reply` ( $p, x, v$ ). If  $o$  is a  $w(x)u$  by  $p$ , then if  $o$  is unmarked, append `buffer-store` ( $p, x, u$ ) to  $\Xi$  and if  $o$  is marked, append `memory-store` ( $p, x, u$ ) to  $\Xi$ .

To complete the proof, we need to show that execution  $\Xi$  complies with  $I$  and  $\mathcal{R}$  of  $M_{TSO}$ . This is achieved with a series of sublemmas and case studies as given in the full version of this paper [8]. ■

Lemmas 1 and 2 together imply:

**Theorem 1.** *A computation is induced by an execution of the system  $(P, J)$  on  $M_{TSO}$  if and only if it is TSO.*

The correctness of our definition of PSO is proved similarly to that for TSO [10].

To show that our definition of RMO is correct requires a different strategy because RMO is not defined as a machine. Rather, RMO is introduced in SPARC v9 [17] and specified by a complicated series of definitions. Again, let  $O$  be all the operations of a computation of system  $(P, J)$ . First a *dependence order* is defined on  $O$  as a three-part restriction of program order. Then *memory order* is defined as a total order on  $O$  that extends the union of a restriction of dependence order and a different restriction of program order. Finally, a *value axiom* determines from the memory order how the value

returned by a read is related to the value written by a write. In the full version of the paper [8], we show that such a memory order exists if and only there is a different order on  $O$  that is both a linearization and that maintains program order on a per-variable basis. Hence, the formal specification of Relaxed Memory Order given by SUN Microsystems [17] is equivalent to that of Definition 1.

## 5 A Small Change with a Huge Consequence

Kohli et al. attempted to provide a non-operational definition for Total Store Ordering [11]. We refer to their definition by TSO-K. TSO-K differs from TSO only in the preceding-read condition, which applies to all reads rather than just foreign reads. The  $K$ -partial program order, denoted  $(O, \xrightarrow{kpo})$ , is the transitive closure of the relation:  $o_1 \xrightarrow{prog} o_2$  intersect  $(o_1, o_2 \in O|x, \text{ for some } x, \text{ or } o_1 \in O|r \text{ or } o_2 \in O|w)$ . Kohli et al. defined the Total Store Ordering model from the point of view of processes but a definition equivalent to theirs is as follows (see [9]):

**Definition 2.** *Let  $O$  be all the operations of a computation  $C$  of a multiprocess system  $(P, J)$ . Then  $C$  is TSO-K if there exists a linearization  $(O, \xrightarrow{L})$  such that  $(O, \xrightarrow{kpo}) \subseteq (O, \xrightarrow{L})$ .*

Definition 2 does not capture TSO semantics. In fact, TSO-K would have substantial implications on the process coordination capabilities of the SPARC architecture (see [10] for proof).

**Theorem 2.** *Let  $\mathcal{P}$  denote a multiprocess program that uses just reads and writes to variables in  $J$ , and let  $\mathcal{P}'$  denote the program obtained from  $\mathcal{P}$  by adding a read invocation of variable  $x$  immediately after every write invocation to  $x$ , for every variable  $x$  in  $\mathcal{P}$ . Then any computation of  $(\mathcal{P}', J)$  on a TSO-K system has exactly the same outcome as some computation of  $(\mathcal{P}, J)$  on a SC system.*

There are several solutions to the critical section problem for SC machines using only read and writes of shared variables. These can be automatically compiled into solutions for machines satisfying TSO-K, using the technique of Theorem 2 [10, 8]. This compilation will not work if the target machine satisfies only TSO because the added reads could be domestic and thus impose no additional ordering constraints. In fact, as will be confirmed in Section 6, there is no solution to the critical section problem using only reads and writes for machines satisfying only TSO.

## 6 Process Coordination for SPARC multiprocessors

### 6.1 Critical Sections Coordination

In the Critical Section Problem (CSP) [15], (also called the Mutual Exclusion Problem) a set of processes coordinate to share a resource. Each process repeatedly cycles through the four procedures  $\langle \text{remainder} \rangle$ ,  $\langle \text{entry} \rangle$ ,  $\langle \text{critical section} \rangle$ ,  $\langle \text{exit} \rangle$  such that

(1) *Mutual Exclusion*: At any time there is at most one process in its <critical section> and (2) *Progress*: If at least one process is in <entry>, then eventually one will be in <critical section>.

We denote a CSP problem by  $CSP(n)$  where  $n \geq 2$  is the number of processes in the system.

**Theorem 3.** *There does not exist an algorithm that solves  $CSP(n)$  for TSO, for any  $n \geq 2$ .*

**Proof Sketch:** Assume that there is an algorithm  $A$  that solves  $CSP(n)$  for TSO for some  $n \geq 2$ . If  $A$  runs with processor  $p$  in <entry> and processor  $q$  in <remainder> and the other processors not participating, then by the Progress property,  $p$  must enter its <critical section> producing a partial computation of the form of Computation 1, where  $\lambda$  denotes the empty sequence and  $o_i^p$  denotes  $p$ 's  $i^{th}$  operation.

**Computation 1**  $\begin{cases} p : o_1^p, o_2^p, \dots, o_k^p & (p \text{ is in its } \langle \text{critical section} \rangle) \\ q : \lambda \end{cases}$

Similarly, if  $A$  runs with  $q$ 's participation only, Progress guarantees that Computation 2 exists.

**Computation 2**  $\begin{cases} p : \lambda \\ q : o_1^q, o_2^q, \dots, o_l^q & (q \text{ is in its } \langle \text{critical section} \rangle) \end{cases}$

Now, computations 1 and 2 are used to construct Computation 3 where both  $p$  and  $q$  are participating, and both are in their <critical section>. By showing that Computation 3 satisfies TSO, we reach a contradiction because Mutual Exclusion is violated.

**Computation 3**  $\begin{cases} p : o_1^p, o_2^p, \dots, o_k^p & (p \text{ is in its } \langle \text{critical section} \rangle) \\ q : o_1^q, o_2^q, \dots, o_l^q & (q \text{ is in its } \langle \text{critical section} \rangle) \end{cases}$

To see that Computation 3 satisfies TSO, imagine a situation where  $p$  and  $q$  enter their critical sections before the contents of their store-buffers are committed to main memory. So, the domestic reads by  $q$  complete at the buffer level, while foreign reads necessarily return initial values.

More formally, construct a sequence,  $S$ , of operations as follows. Initially,  $S = \lambda$ . Set sequence  $Q$  to be  $q$ 's computation  $\langle o_1^q, o_2^q, \dots, o_l^q \rangle$ . Examine each  $o_i^q$  in  $Q$  in order from  $i = 1$  to  $l$ . If  $o_i^q$  is a foreign read, append  $o_i^q$  to  $S$  and remove it from  $Q$ . When there are no foreign reads left in  $Q$ , append to  $S$   $p$ 's computation  $\langle o_1^p, o_2^p, \dots, o_k^p \rangle$ . Finally, append  $Q$  (with foreign reads removed) to  $S$ .

Define  $(O, \xrightarrow{L})$  to be  $S$ .  $(O, \xrightarrow{L})$  consists of three segments. The first consists entirely of foreign reads by  $q$ , the second consists entirely of  $p$ 's computation, and the third consists entirely of  $q$ 's computation minus operations in the first segment.

It is straightforward to check that  $(O, \xrightarrow{L})$  is a linearization. To see that  $(O, \xrightarrow{L})$  satisfies TSO, note first that  $(O|p, \xrightarrow{L}) = (O|p, \xrightarrow{prog})$ . Consequently,  $(O|p, \xrightarrow{tso}) \subseteq (O|p, \xrightarrow{L})$ . Second, program order is maintained in the first segment by construction and also in the third segment. Finally, each read moved to the first segment is foreign and returns the initial value. Therefore, such a read is not preceded in  $q$ 's computation by any writes to the same variable. So the moved reads do not violate  $(O, \xrightarrow{tso})$ . Therefore,  $(O, \xrightarrow{tso}) \subseteq (O, \xrightarrow{L})$ . Thus, Computation 3 satisfies TSO. ■

Since the preceding impossibility result made no assumption about fairness or about size of variables, it implies impossibility even of unfair solutions or solutions using unbounded variables.

## 6.2 Producer/Consumer Coordination

Producer/Consumer [2] objects are frequently used for process coordination. The producer (respectively, consumer) is a process that repeatedly cycles through the procedures  $\langle \text{entry} \rangle$ ,  $\langle \text{producing} \rangle$  (respectively,  $\langle \text{consuming} \rangle$ ),  $\langle \text{exit} \rangle$ . Let  $m$  and  $n$  be, respectively, the number of producer and consumer processes. We distinguish two problems the solution requirements of which vary: the *producer/consumer set* problem, denoted  $P_m C_n$ -set, and the *producer/consumer queue* problem, denoted  $P_m C_n$ -queue. A solution to  $P_m C_n$ -set must satisfy (1) *Safety*: Every produced item is consumed exactly once, and every consumed item is produced exactly once, and (2) *Progress*: If a producer (respectively consumer) is in  $\langle \text{entry} \rangle$ , then it will eventually be in  $\langle \text{producing} \rangle$  (respectively  $\langle \text{consuming} \rangle$ ) and subsequently in  $\langle \text{exit} \rangle$ . A solution to  $P_m C_n$ -queue must satisfy Safety, Progress, and (3) *Order*: Consumption order must respect the production order, where these orders are defined as follows. Item  $x$  *precedes in production order* item  $y$  if the production of  $x$  completes before the production of  $y$  begins. Consumption order is defined similarly.

**Impossibilities:** The general queue problems  $P_m C_n$ -queue,  $P_1 C_n$ -queue, or  $P_m C_1$ -queue are unsolvable for TSO, PSO, and RMO without using explicit synchronization. The proof of the following theorem is very similar to that of Theorem 3.

**Theorem 4.** *There does not exist an algorithm that solves  $P_m C_n$ -queue or  $P_m C_n$ -queue for TSO when  $m + n \geq 3$ .*

**Possibilities:** Proofs of the following theorems are in the full version of the paper [8].

**Theorem 5.** *There is a single-writer solution for  $P_1 C_1$ -queue for TSO (which fails for PSO and hence RMO).*

**Theorem 6.** *There is a multi-writer solution for  $P_1 C_1$ -queue for RMO (and hence for PSO and TSO).*

**Theorem 7.**  *$P_m C_n$ -set is solvable for RMO (and hence for TSO or PSO).*

## 7 SPARC Models versus other Consistency Models

In the full paper [8] we compare TSO, PSO and RMO with other known memory models including Weak Ordering (WO) [3], Java [5, 7], Causal Consistency (CC) [1], Processor Consistency (PC-G) [6], and Pipelined-Random Access Machine (P-RAM) [13], Coherence [3], and Sequential Consistency (SC) [12]. A summary of these comparisons is given in Figure 3. In this figure, an arrow from model *A* to model *B* means that the constraints of *A* imply those of *B*.

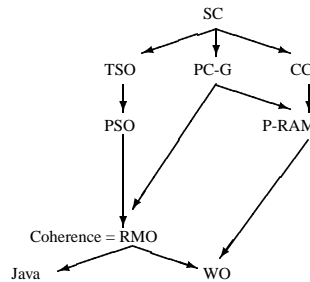


Fig. 3. Relationships between SPARC models and other models

## References

1. M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementations, and programming. *Distributed Computing*, 9:37–49, 1995.
2. E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965. Reprinted in [4].
3. M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Int'l Symp. on Computer Architecture*, pages 434–442, June 1986.
4. F. Genuys, editor. *Programming Languages*. Academic Press, 1968.
5. A. Gontmakher and A. Schuster. Characterizations of Java memory behavior. In *Proc. 12th Int'l Parallel Processing Symp.*, April 1998.
6. J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.
7. L. Higham and J. Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. 12th Int'l Symp. on Distributed Computing, Lecture Notes in Computer Science volume 1499*, pages 201–215, September 1998.
8. L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. Technical report, Department of Computer Science, The University of Calgary, 2000.
9. J. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems*. Ph.D. dissertation, Department of Computer Science, The University of Calgary, January 2000.
10. J. Kawash and L. Higham. Memory consistency and process coordination for SPARC v8 multiprocessors. Technical Report 99/646/09, Department of Computer Science, The University of Calgary, December 1999. A brief announcement appeared in *Proc. 19th ACM Symp. on Principles of Distributed Computing*, page 335, July 2000.
11. P. Kohli, G. Neiger, and M. Ahamad. A characterization of scalable shared memories. In *Proc. 1993 Int'l Conf. on Parallel Processing*, August 1993.
12. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

13. R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.
14. S. Park and D. L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers*, 48(2):227–235, February 1999.
15. M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.
16. SPARC International, Inc. *The SPARC Architecture Manual version 8*. Prentice-Hall, 1992.
17. D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual version 9*. Prentice-Hall, 1994.