Specifying Memory Consistency of Write Buffer Multiprocessors

Lisa Higham[(1)] LillAnne Jackson[(1)] Jalal Kawash[(1),(2)]
higham@cpsc.ucalgary.ca jackson@cpsc.ucalgary.ca jkawash@ausharjah.edu

[(1)]Department of Computer Science
The University of Calgary, Calgary, Canada
[(2)]Department of Computer Science
American University of Sharjah, Sharjah, UAE

Abstract

Write buffering is one of many successful mechanisms that improves the performance and scalability of multiprocessors. However, it leads to more complex memory system behavior, which cannot be described using intuitive consistency models, such as sequential consistency. It is crucial to provide programmers with a specification of the exact behavior of such complex memories. This paper presents a uniform framework for describing systems at different levels of abstraction and proving their equivalence. The framework is used to derive and prove correct specifications in terms of program-level instructions of the SPARC total store order and partial store order memories.

The framework is also used to examine the SPARC relaxed memory order. We show that it is not a memory consistency model that corresponds to any implementation on a multiprocessor that uses write-buffers, even though we suspect that the SPARC Version 9 specification of relaxed memory order was intended to capture a general write-buffer architecture. The same techniques can be used to show that coherence does not correspond to a write-buffer architecture. A corollary is that any implementation of Alpha consistency using write-buffers cannot produce all possible Alpha computations. That is, there are some computations that satisfy the Alpha specification but cannot occur in the given write-buffer implementation.

Keywords: Memory consistency framework, Write-buffer architectures, SPARC multiprocessors, Total store order, Partial store order, Relaxed memory order, Sequential consistency, Alpha, Coherence

1 Introduction

Distributed algorithm designers typically assume strong consistency models such as sequential consistency and linearizability [Lam79a, HW90]. Multiprocessor architectures, however, incorporate mechanisms such as write-buffers to improve the performance and scalability. Such mechanisms lead to the relaxation of the resulting memory consistency model. These weakened consistency guarantees make it challenging for a programmer to be certain of the possible program outcomes and to implement correct and efficient distributed algorithms. Some attempted specifications of the behavior of such complex memory systems resulted in incomplete or incorrect conclusions. Thus, such specifications must be shown to exactly capture the operational behavior of the underlying memory architecture.

This paper presents a framework that allows us to describe systems at various levels of abstraction, ranging from operational or architectural hardware functionality, to object oriented or transaction based abstractions. The uniform framework used for such descriptions facilitates the comparison of different models of consistency, so that claims of the relative strength of models can be stated precisely and proven correct. In particular it provides the setting in which to prove that a non-operational description of a multiprocessor system captures exactly the operational behaviour of that system. The uniformity of the descriptions at various levels of abstraction means that proofs of correct implementations of abstract systems on systems with finer granularity of description can be composed. Starting from a highly abstract description, a sequence of provably correct implementations onto successively lower levels of abstraction can be constructed within the framework, eventually arriving at the actual operational machine. Thus, the framework can be seen as providing a way to generalize the well established notions of levels of abstraction and proofs of correctness of implementations, to include arbitrary memory consistency models.

The framework is used to describe, at an operational level, the execution of programs on possible write-buffer multiprocessor architectures. Three different write-buffer machines are considered. The first represents the "bare minimum" write-buffer machine capturing the very basic behavior of write-buffers. The other two successively stronger machines capture the operational behavior of SPARC's partial and total store order machines [SPA92, WG94]. Using the framework, three variants of the outcomes of executions of programs on these machines are specified, each meeting additional operational constraints of the machine.

The same framework is used to provide two more abstract (non-operational) memory consistency models, corresponding to the SPARC's total and partial store order machines. These higher-level models constitute the programmer's view of the behavior of multiprocess programs on the respective machines. For each of the two pairs, the operational and non-operational descriptions are proven to be equivalent.

The SPARC version 9 manual [WG94] specifies a third memory consistency model called relaxed memory order, which seems to be intended to capture the executions of a still more permissive write-buffer architecture. Exploiting our framework, we show, however, that relaxed memory order does not correspond to any implementation on a multiprocessor that uses write-buffers. In fact, we prove that coherence [DSB86, Goo89] or any weaker model does not correspond to a write-buffer architecture. Since it has been shown that Alpha consistency [Cor98] is coherence combined with some program dependencies [AF94], a corollary is that any implementation of Alpha consistency using write-buffers cannot produce all possible Alpha computations. Thus any implementation of Alpha consistency using write-buffers must produce a memory consistency model strictly stronger than that specified by the Alpha documentation [Cor98], even though this documentation suggests such an implementation.

Software system designers require an unambiguous description of the memory consistency conditions at the level of the operations used in their programs. Typically, hardware architecture manuals use natural language or axiomatic specifications to describe how the hardware operates [SPA92, WG94, Int02, Int97]. These descriptions are not programmer centric and may contain ambiguities. They can lead to incorrect or inefficient programs and they are complex to work with. Frameworks have been defined to provide a way to formalize and unify descriptions of memory consistency models, and to help us reason about them. One of the earliest frameworks, an axiomatic one by Hoare [Hoa72], is difficult to use to prove correctness even for simple algorithms. Owicki and Gries [OG76] provide strengthened axioms and used them to provide proofs of coordination algorithms in a parallel programming language. Misra [Mis86] presented a framework and applies it to a set of register axioms. In each case, the framework is customized for a particular level of system design (parallel programs or hardware registers). In 1979, Lamport [Lam79b] provided an axiomatic system and used it as the basis for descriptions and proofs of distributed system features [Lam97, Lam86b, Lam86a]. In his extension of these axioms, Anger [Ang89] states that "The

method of analysis for high-level system descriptions is the same as for low-level operation descriptions."

The framework developed by Attiya, Friedman *et. al.* [ACFW98, AF92, Fri95] is used to describe coordination algorithms and provide transformations of programs written assuming one memory consistency model to new programs intended to be run under different consistency assumptions. Ahamad *et. al.* [ABJ+93, ANB+95, KNA93] present a framework that is used to define causal consistency and to re-state some memory consistency definitions, such as processor consistency and pipelined-RAM. Gibbons and Merrit [GM92] present a framework that uses an I/O automaton to represent architectural assumptions and a memory consistency model that restricts its actions.

Our framework uses ideas from much of this previous work. It supports descriptions at various level of abstraction, as does Lamport's. Transformations between different memory consistency models are facilitated by the framework, as in the work of Attiya and Friedman. The descriptive style provided by the framework is general, but is is similar to that of several earlier works including that of Ahamad *et. al.*.

Capturing the semantics of the SPARC memory consistency models simply and precisely has proven to be surprisingly tricky. One earlier attempt to define total store order [KNA93] resulted in a definition that is stronger than what the architecture provides and leads to erroneous conclusions about the coordination capabilities under total store order consistency. In fact, any program using only read and write operations on variables that is correct for sequential consistency can be compiled into an equivalent program with only read/write operations that is correct for this erroneous description of total store order [HK00]. Thus, for example, there is a solution to mutual exclusion that uses only variables for this version of total store order. Our earlier work proved, to the contrary, that variables are insufficient to solve the mutual exclusion problem on a total store order machine [HK97, Kaw00].

The simplicity of the SPARC memory consistency descriptions provided in this paper gives programmers an improved tool for reasoning about the outcomes of their programs. It facilitates the comparison of the SPARC models to each other and to proposed consistency models including processor consistency [Goo89], causal consistency [ANB+95], and Java consistency [GS00, HK98], and aids the development of verification tools [PD99].

The rest of the paper is organized as follows. Section 2 describes the framework that is used throughout the paper. In Section 3, the framework is used to specify three write-buffer machines, two of which capture the familiar total-store-order and partial-store-order semantics of a multiprocessor with write-buffers. Section 4 defines two non-operational memory consistency specifications. It is then proved that the total store order and partial store order machines implement exactly these specifications, establishing that these two non-operational specifications are correct. Section 5 establishes that these operational and non-operational models are equivalent to the corresponding consistency models described in the SPARC manuals. Section 6 shows that the memory consistency models defined by the SPARC relaxed memory order, Alpha consistency, and coherence cannot be described as arising from any write-buffer machine. Section 7 summarizes and concludes.

2 Framework

2.1 Describing systems

A multiprocess system can be modeled as a collection of *programs* operating on a collection of *shared data objects* under some partial order constraints called a *memory consistency model*. In this section we specify each of these components. Three running examples are provided and will be used in subsequent sections of the paper.

Shared data objects: A shared data object might be described by providing an object's initial state, the operations that can be applied to it and the change of state and response that results from each applicable operation. This gives rise to a set of allowable sequences of operations for each such object. So, we specify a *shared data object* to be a set of sequences of operations. An *operation* has the form *act(obj,in,out)* where 'act' is an action with input parameters in the list 'in' applied to object 'obj' and that returns the output values in the list 'out'. If 'in' is non-empty the operation is a *state-change operation* and if 'out' is non-empty the operation is an *output-generating operation*. An operation could be both a state-change and an output-generating operation. An operation act(obj,in,out) has two components; its *invocation* is *act(obj,in)* and its *response* is *out*.

An arbitrary sequence of operations applied to object X is *valid for X* if and only if it is in the specification of X. An arbitrary sequence S of operations (applied to possibly several objects) is *valid* if and only if, for each object X, the subsequence of S consisting of exactly those operations applied to X is valid for X.

In this paper, we consider three types of objects: read/write objects (which we call variables), set objects and list objects. Throughout, we assume that for each state-change operation the input parameter has a distinct value. Also, for this paper, any non-empty input or output list contains only one item and is therefore abbreviated by omitting list delimiters. An empty input or output list is denoted by λ. An *operation type* is just an operation that has one or more parameters chosen from some given set. The set of these possible parameters is indicated by a '$\cdot$' in the parameter list. For example the notation write($x,\cdot,\lambda$) denotes the set of write actions on object x, or (depending on context) any element of that set.

Example 1a – variable: A *variable*, x, is the set of sequences over operations of the type write($x,\cdot,\lambda$) and read($x,\lambda,\cdot$) that satisfy the validity condition:

> The output value returned by each read operation is the same as the input value written by the most recent preceding write operation in the sequence, if such a write exists, and is $\perp$ otherwise.

Example 2a – set object: A *set object* s, is the set of sequences over operations of the type insert($s,\cdot,\lambda$), delete($s,\cdot,\lambda$), and member($s,\lambda,\cdot$) that satisfy the validity condition:

> For any member operation on set object, s, if it returns a value $\rho \neq \perp$ then there is a preceding insert(s,ρ,λ) operation[1]. Furthermore, between this insert and the member there is no delete(s,ρ,λ). Otherwise, it returns $\perp$.

Example 3a – list object: A *list object*, l, is the set of sequences over operations of the type append($l,\cdot,\lambda$), delete($l,\cdot,\lambda$), and last($l,\lambda,\cdot$) that satisfy the validity condition:

> For any last operation on list object, l, if it returns a value $\rho \neq \perp$ then the most recent preceding append operation is append(l,ρ,λ). Furthermore, between this last and the append there is no delete(l,ρ,λ). Otherwise, it returns $\perp$. [2]

Definitions of objects typically constrain the output of output-generating operations to be related to the input of some preceding state-change operation. For variables, a read operation returns the value of a preceding write operation; for set objects, member returns the value of a preceding insert operation; and for

[1] Recall that every state-change operation has a distinct value.

[2] Both list and set objects could be more tightly constrained. For example, it would be natural to require of list objects that only items previously appended can be deleted. The weaker definition chosen here suffices for this paper because in our implementations, set and list objects will be used in ways that force such extra requirements.

list objects last returns the value of the most recent preceding append operation. In each case, the output-generating operation and the unique related state-change operation are said to be *causally related*.

Henceforth, we abbreviate common operations and operation invocations that have either input or output parameters but not both, by simply omitting the missing parameter. For example, the operation write(x,ρ,λ) is abbreviated write(x,ρ), last(l,λ,ρ) is abbreviated last(l,ρ), and the operation invocation read(x,λ) is abbreviated read(x).

Programs and multiprocesses: An *individual program* in a multiprocess system is sequential computer code consisting of local operation invocations, local computation, control structures, and operation invocations on shared objects. A collection of individual programs is a *multiprogram*. Let P be a multiprogram whose non-local operation invocations are applied to shared objects in J. Then the pair (P,J) is called a *multiprocess* and P is *compatible* with J.

Example 1b – variable multiprogram: Let x and y be variables, and let $\{p_1, p_2\}$ be the multiprogram:

$$\begin{cases} p_1 : \text{write}(x,4)\ \ \text{read}(y)\ \ \text{write}(y,5)\ \ \text{read}(x) \\ p_2 : \text{write}(x,6)\ \ \text{read}(x)\ \ \text{write}(y,2)\ \ \text{read}(y) \end{cases}$$

Then $(\{p_1, p_2\}, \{x,y\})$ is a (variable) multiprocess and $\{p_1, p_2\}$ is compatible with $\{x,y\}$.

Example 2b – set object multiprogram: Let s be a set object, and let $\{p_1, p_2\}$ be the multiprogram:

$$\begin{cases} p_1 : \text{insert}(s,4)\ \ \text{insert}(s,6)\ \ \text{member}(s) \\ p_2 : \text{insert}(s,7)\ \ \text{delete}(s,6)\ \ \text{member}(s)\ \ \text{delete}(s,4) \end{cases}$$

Then $(\{p_1, p_2\}, \{s\})$ is a (set object) multiprocess and $\{p_1, p_2\}$ is compatible with $\{s\}$.

Example 3 – list object multiprogram: Let l be a list object, and let $\{p_1, p_2\}$ be the multiprogram:

$$\begin{cases} p_1 : \text{append}(l,4)\ \ \text{append}(l,6)\ \ \text{last}(l) \\ p_2 : \text{append}(l,7)\ \ \text{delete}(l,6)\ \ \text{last}(l)\ \ \text{delete}(l,4) \end{cases}$$

Then $(\{p_1, p_2\}, \{l\})$ is a (list object) multiprocess and $\{p_1, p_2\}$ is compatible with $\{l\}$.

To highlight the association of operations with their invoking programs, 'act' is subscripted with the program identifier when required. For instance, the notation write$_p(x,\rho)$ emphasizes that this write operation is invoked by individual program p.

Memory consistency models: The execution of a sequentially consistent [Lam79a] multiprocess can be modeled by a single valid sequence of operations that arises from interleaving the operation invocations generated by the individual programs of the multiprocess. In weaker memory consistency models, no such straightforward interleaving is guaranteed. To model these more general settings, we first define a computation.

Each individual program's *individual computation* is a sequence of operations. The order of the operations in this sequence is called *program order* and is the same as the order in which the corresponding operation invocations appear in the individual program. A *(system) computation* of a multiprocess (P,J) is a collection of individual computations, one for each individual program, $p \in P$.

Example 1c – read/write computation: C_1 is a computation of the read/write multiprocess in Example 1b.

$$C_1 \begin{cases} p_1 : \text{write}(x,4)\ \ \text{read}(y,2)\ \ \text{write}(y,5)\ \ \text{read}(x,6) \\ p_2 : \text{write}(x,6)\ \ \text{read}(x,4)\ \ \text{write}(y,2)\ \ \text{read}(y,5) \end{cases}$$

Example 2c – set object computation: C_2 is a computation of the set object multiprocess in Example 2b.

$$C_2 \begin{cases} p_1 : \text{insert}(s,4)\ \ \text{insert}(s,6)\ \ \text{member}(s,4) \\ p_2 : \text{insert}(s,7)\ \ \text{delete}(s,6)\ \ \text{member}(s,7)\ \ \text{delete}(s,4) \end{cases}$$

Example 3c – list object computation: C_3 is a computation of the list object multiprocess in Example 3b.

$$C_3 \begin{cases} p_1 : \text{append}(l,4)\ \ \text{append}(l,6)\ \ \text{last}(l,4) \\ p_2 : \text{append}(l,7)\ \ \text{delete}(l,6)\ \ \text{last}(l,7)\ \ \text{delete}(l,4) \end{cases}$$

Notice that the response components of the operations in a computation are unconstrained by the preceding definition. Rather, the possible responses are determined by the architecture, which we model as a set of memory consistency constraints. Thus, a *memory consistency model* is a set of partial order constraints on the operations of a computation. These partial orders are defined on subsets of all the operations. For example, Lamport's sequential consistency [Lam79a] requires: "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."

Let O be all the operations of a computation C of a multiprocess (P,J). The program order on O is a partial order denoted $(O, \xrightarrow{prog})$. In our framework, sequential consistency can be defined as follows.

Sequential consistency: C satisfies *sequential consistency* if there is a valid total order $(O, \xrightarrow{sc})$ such that $(O, \xrightarrow{prog}) \subseteq (O, \xrightarrow{sc})$.

Several partial and total orders will be defined in this paper. For any partial order, $(O, \longrightarrow)$, the notation $o_1 \longrightarrow o_2$ means $(o_1, o_2) \in (O, \longrightarrow)$. Also, $\longrightarrow$ abbreviates $(O, \longrightarrow)$ when the set O is clear. If $(O, \xrightarrow{L})$ is a total order, there is exactly one sequence of the elements of O that maintains this order; we denote that sequence by L.

Example 1d – sequential consistency on a variable multiprocess: Computation C_1 (of Example 1c) is not sequentially consistent because it is not possible to create a total order that both maintains program order and is valid. However, if $\text{read}_{p_1}(x,6)$ in C_1 is replaced by $\text{read}_{p_1}(x,4)$ the resulting computation is sequentially consistent. For example one valid total order that preserves program order is: $\text{write}_{p_2}(x,6)$ $\text{write}_{p_1}(x,4)$ $\text{read}_{p_2}(x,4)$ $\text{write}_{p_2}(y,2)$ $\text{read}_{p_1}(y,2)$ $\text{write}_{p_1}(y,5)$ $\text{read}_{p_1}(x,4)$ $\text{read}_{p_2}(y,5)$.

Example 2d – sequential consistency on a set object multiprocess: Computation C_2 (of Example 2c) is sequentially consistent as shown by the following valid total order for set objects: $\text{insert}_{p_1}(s,4)$ $\text{insert}_{p_2}(s,7)$ $\text{insert}_{p_1}(s,6)$ $\text{member}_{p_1}(s,4)$ $\text{delete}_{p_2}(s,6)$ $\text{member}_{p_2}(s,7)$ $\text{delete}_{p_2}(s,4)$.

Example 3d – sequential consistency on a list object multiprocess: Computation C_3 (of Example 3c) is not sequentially consistent because there is no total order that maintains program order and ensures that $\text{last}_{p_1}(l,4)$ is valid.

Partial order constraints that are weaker than sequential consistency constitute a *weak* memory consistency model.

Let J be a set of objects and P a multiprogram compatible with J, and let M be a memory consistency model. Then the triple (P,J,M) is called a *multiprocess system* and the couple (J,M) is called a *platform*. The *computations of a multiprocess system* (P,J,M) are all the computations of the multiprocess (P,J) that satisfy the memory consistency constraints M.

2.2 Comparing and implementing systems

Transformations between multiprocesses provide our main tool for comparing systems, and for developing abstract descriptions of the memory consistency models that arise from concrete or operational level executions of multiprograms. One multiprocess, called the *specified multiprocess* (with corresponding *specified* objects and multiprograms) is transformed into the *target multiprocess* (with corresponding *target* objects and multiprograms). An operation on a specified object is *transformed* to the target objects by providing a subroutine for the operation's invocation where this subroutine uses only operation invocations on the target objects. If the specified operation is output-generating, then the subroutine must return a value of the same type as this output. An *object is transformed* to the target object(s) by transforming each of its operations. A transformation of each of the objects of a specified multiprocess to objects of the target multiprocess can be naturally extended to a *program transformation* by replacing each operation invocation in the specified multiprogram with the subroutine for that operation invocation. The transformed multiprogram together with the target objects comprise the *transformed multiprocess.*

The transformed multiprocess and a target memory consistency model (called the *target system*) gives rise to a collection of computations — exactly those that arise from the transformed multiprogram interacting with the target objects and satisfying the target memory consistency constraints. Any such computation can be *interpreted* as a computation of the specified multiprogram by attaching to each operation invocation of the specified multiprogram, the value returned by the corresponding subroutine. Thus, the set of computations of the target system provides a set of interpreted computations of the specified multiprocess.

Another way to associate a set of computations with the specified multiprocess is to specify a memory consistency model and consider the set of computations of this specified system.

The relationship between these two sets of computations of the specified system determines whether or not we consider the transformation to be an implementation. A transformation of a specified multiprogram will be called an implementation of the specified system, if, informally (in Figure 1) the set $\mathcal{C}'$ of computa-

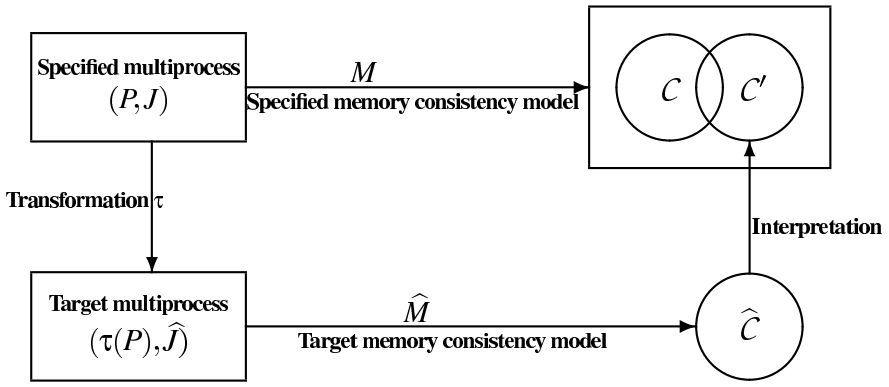

Figure 1: Program transformation and computation interpretation.
If $\mathcal{C}' \subseteq \mathcal{C}$, then the transformation is an implementation.

tions produced by traveling the long way around is a non-empty subset of the computations C allowed by the specified system. More precisely, let C be the set of computations of a specified multiprocess system (P,J,M). Let τ be a transformation from the objects in J to the objects in $\widehat{J}$ and denote by $\tau(P)$ the multiprogram constructed from the extension of τ to apply to multiprogram P. Let $\widehat{D}$ be any computation of system $(\tau(P),\widehat{J},\widehat{M})$ and let D' be the interpretation of $\widehat{D}$ for (P,J). Then the transformation τ *implements the system* (P,J,M) *on the platform* $(\widehat{J},\widehat{M})$ if $D' \in C$, for any such D'. The transformation τ *exactly implements the system* (P,J,M) *on the platform* $(\widehat{J},\widehat{M})$ if τ implements (P,J,M) on $(\widehat{J},\widehat{M})$ and, for any $D \in C$ there is a computation $\widehat{D}$ of $(\tau(P),\widehat{J},\widehat{M})$ whose interpretation is D. A transformation is a *compiler* (respectively, an *exact compiler*) from platform (J,M) to platform $(\widehat{J},\widehat{M})$ if, for any multiprogram P compatible with J, the transformation implements (respectively, exactly implements) the system (P,J,M) on the platform $(\widehat{J},\widehat{M})$.[3]

Throughout this paper, if a transformation τ applied to operation o produces a sub-routine with operations $\widehat{o_1}$, $\widehat{o_2}$, ..., $\widehat{o_l}$, then for any i and j in $\{1,\ldots,l\}$, o and $\widehat{o_i}$ are called *corresponding* operations and $\widehat{o_i}$ and $\widehat{o_j}$ are called *matching operations*.

3 Write-Buffer Architectures

One common multiprocessor architecture, such as the SPARC multiprocessor, associates a write-buffer with each processor as shown in Figure 2. The main memory is single ported with a non-deterministic switch

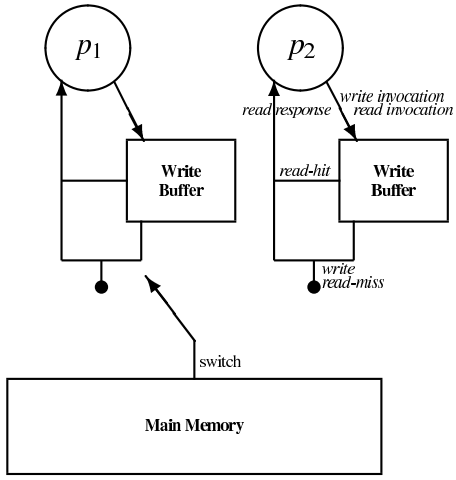

Figure 2: Write-Buffer Architecture (for 2-processors)

providing one memory access at a time. Each write-buffer operates in parallel with the processor.

We contend that even the most general variant of any write-buffer machine will execute as follows. When a processor performs a write it need not wait for it to be committed to main memory. Instead, it is stored in the write-buffer, which is responsible for committing pending writes to main memory. When a read is issued by a processor, the processor's associated write-buffer is checked for pending writes to the same location. If there is any such write, the value "to be written" by some such write to that location is returned. In this case, the read completes without accessing main memory. Otherwise, the read accesses main memory and returns the value of the location in main memory. Notice that this basic machine has quite

[3]Lamport [Lam86b] similarly distinguishes between prescriptive and restrictive definitions of correctness of implementations.

weak operational constraints. A read action applied to a location can return any value for that location that is not yet committed. Furthermore, operations need not be blocking; once the buffer action is complete a process can invoke its next operation in program order. However, accesses to any individual location in main memory by any one processor are via a FIFO channel. Therefore, for each location, pending memory write and read operations of a processor to that location are performed in the same order as they are issued. This means that although the write buffer need not be emptied in FIFO order, writes by a single processor to a single location will be committed to main memory in FIFO order. Furthermore, a read that failed at the write buffer, cannot skip over an earlier write that is being committed to main memory by the same processor to the same location.

Two more restrictive write-buffer behaviours are distinguished by 1) constraining the order in which pending main memory operations are performed; 2) constraining which value of a location that is in the write-buffer can be returned by a read action; and 3) requiring the processor to block on reads.

A partial store order write-buffer machine, which is modeled after a SPARC machine [SPA92], adds two constraints to the basic write-buffer machine: 1) when the write-buffer contains pending writes to a location being read, the value of the most recent such write is returned; and 2) when the write-buffer does not contain any pending writes to a location being read, the processor blocks until the read is complete.

A total store order write-buffer machine, also modeled after a SPARC machine, further constrains the partial store order write-buffer machine by requiring that all pending writes from a write buffer to main memory (rather than just those to the same location) must be performed in FIFO order.

Throughout we address only read and write operations to variables, even though multiprocessors support other operations that affect shared memory. Complicated memory consistency behaviour arises, however, because read and write operations are executed as a sequence of (sometimes, non-blocking) steps. Operations such as various read-modify-write and memory barrier operations impose stronger coordination than reads and writes of variables, and hence they limit or eliminate these complicated scenarios. Thus, they can be easily added to all the systems considered here [Kaw00]. Since it is most challenging to pin down the memory consistency constraints of the weak operations, we focus on them in this paper. We are also motivated to minimize the use of strong synchronization primitives since they negatively impact the system performance. So this paper focuses on specifying the behaviours of these systems when they avoid the use of the stronger operations.

3.1 Basic write-buffer system

The informal descriptions of the behaviour of each write-buffer machine can be converted to a precise definition of a system, using the framework of Section 2. For the remainder of this paper, J is a set of (specified) variables and $P = \{p_1, \ldots, p_n\}$ is a multiprogram compatible with J. A target write-buffer system associated with multiprocess (P,J) is defined by specifying each of its components.

The *basic write-buffer system* $(\widehat{P},\widehat{J},\widehat{WB})$ *associated with multiprocess* (P,J) is defined as follows.

Objects: For each variable in J there is a corresponding main-memory variable and there is a set object for that variable in each processor's write-buffer. More precisely, for each $x \in J$, associate with x a variable $\widehat{x}$, and for each pair $x \in J$ and $i \in \{1, \cdots, n\}$, associate a set object $\widehat{x_i}$. The set of objects $\widehat{J}$ is:

$$\widehat{J} = \bigcup_{x \in J} (\{\widehat{x}\} \cup \{\widehat{x_1}, \ldots, \widehat{x_n}\})$$

Programs: To define the transformation γ from the read and write operations on the variables in J to operations on the objects in $\widehat{J}$, it is convenient to extend the definition of an operation so that it is applicable to more than one object. A *compound operation* has the form *act(objs,in,out)* where 'act' is an action

applied to the collection of objects 'objs' with input parameters in the list 'in' returning the output values in the list 'out' and which is defined as an indivisible sequence of operations. For set object or list object $\widehat{x_i}$ and variable $\widehat{x}$, define the compound operation move($\{\widehat{x_i},\widehat{x}\}$,$v$,$\lambda$) (abbreviated move($\{\widehat{x_i},\widehat{x}\},v$)) to be the indivisible sequence of operations: delete($\widehat{x_i},v$),write($\widehat{x},v$). A sequence of operations on set objects and variables (or on list objects and variables) that includes move operations is *valid* if and only if the sequence modified by replacing each move with the delete,write sequence that defines the move is valid.

To distinguish between read operations on variables in J, and read operations on the variables in $\widehat{J}$, the latter are renamed mem-read. The transformation γ from J to $\widehat{J}$ is defined by:

define $\gamma(\text{read}_{p_i}(x))$ =
  $v \leftarrow \text{member}(\widehat{x_i})$
  if $(v = \perp)$ **then**
    $v \leftarrow \text{mem-read}(\widehat{x})$
  return v

define $\gamma(\text{write}_{p_i}(x,v))$ =
  insert$(\widehat{x_i},v)$
  move$(\{\widehat{x_i},\widehat{x}\},v)$

The transformation γ is used to transform each write and each read operation invocation in P yielding a transformed multiprogram $\widehat{P} = \gamma(P)$. Clearly, $\widehat{P}$ is compatible with $\widehat{J}$. The program order of operations in the transformed muliprogram $\gamma(P)$ is denoted $\xrightarrow{\widehat{prog}}$.

Memory Consistency: Let $\widehat{O}$ be all the operations of a computation $\widehat{C}$ of $(\gamma(P),\widehat{J})$. Subsets of $\widehat{O}$ are denoted by $\widehat{O}|_Q$ where Q is a collection of operation types, a process or an object. For example, $\widehat{O}|_{\text{insert}\cup\text{member}}$ is the set of all the insert and member operations in $\widehat{O}$ and $\widehat{O}|_{\widehat{x}}$ is the subset of $\widehat{O}$ consisting of all the operations on object $\widehat{x}$. The notation is often combined to produce an intersection of these subsets. For example, $\widehat{O}|_{\text{mem-read}}|_{\widehat{x}}$ is the subset of $\widehat{O}$ that contains all the mem-read operations applied to variable $\widehat{x}$. Recall that o and $\widehat{o_i}$ correspond if $\widehat{o_i}$ is in the program $\gamma(o)$ and that $\widehat{o_i}$ and $\widehat{o_j}$ are matching operations if they correspond to the same operation. Define three partial orders on $\widehat{O}$.

matching-ops: $(\widehat{o_1},\widehat{o_2}) \in$ *matching-ops* if and only if $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ and $\widehat{o_1}$, $\widehat{o_2}$ are matching operations.

buffer-sets: $(\widehat{o_1},\widehat{o_2}) \in$ *buffer-sets* if and only if $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ and $\widehat{o_1},\widehat{o_2} \in \widehat{O}|_{\text{insert}\cup\text{member}}$.

FIFO-per-location-memory: $(\widehat{o_1},\widehat{o_2}) \in$ *FIFO-per-location-memory* if and only if $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ and $\widehat{o_1},\widehat{o_2} \in \widehat{O}|_{\text{mem-read}\cup\text{move}}|_{\widehat{x}}$, for some variable $\widehat{x} \in \widehat{J}$.

Informally, matching-ops order ensures that each read and write is implemented according to the definition of its transformation; buffer-sets order ensures that each operation appears to be invoked in program order (though they need not complete in that order); FIFO-per-location-memory ensures that move and mem-read operations by the same individual program to the same location remain in program order. Thus these orders capture the informal description of the behaviour of the most general write-buffer machine.

basic write-buffer consistency: Computation $\widehat{C}$ satisfies *basic write-buffer consistency* (abbreviated $\widehat{WB}$) if there is a valid total order of all the operations $\widehat{O}$ of $\widehat{C}$ that preserves matching-ops order, buffer-sets order, and FIFO-per-location-memory order.

3.2 Partial and total store order write-buffer systems

Partial store order and total store order write-buffer systems are defined similarly to the basic write-buffer system except that $\widehat{J}$ contains list objects (instead of set objects). The *partial store order write-buffer system*

$(\widehat{P},\widehat{J},\widehat{PSO})$ and the *total store order write-buffer system* $(\widehat{P},\widehat{J},\widehat{TSO})$ *associated with multiprocess* (P,J) are defined as follows. They differ only in the memory consistency component.

Objects: For each variable in J there is a corresponding main-memory variable and there is a list object for that variable in each processor's write-buffer. More precisely, for each $x \in J$, associate with x a variable $\widehat{x}$, and for each pair $x \in J$ and $i \in \{1,\cdots,n\}$, associate a list object $\widehat{x}_i$. The set of objects $\widehat{J}$ is:

$$\widehat{J} = \bigcup_{x \in J} (\{\widehat{x}\} \cup \{\widehat{x}_1,\ldots,\widehat{x}_n\})$$

Programs: The transformation τ from J to $\widehat{J}$ is defined by:

define $\tau(\text{read}_{p_i}(x))$ =
  $v \leftarrow \text{last}(\widehat{x}_i)$
  if $(v = \perp)$ **then**
    $v \leftarrow \text{mem-read}(\widehat{x})$
  return v

define $\tau(\text{write}_{p_i}(x,v))$ =
  $\text{append}(\widehat{x}_i, v)$
  $\text{move}(\{\widehat{x}_i,\widehat{x}\}, v)$

The transformation τ is used to transform each write and each read operation invocation in P yielding a transformed multiprogram $\widehat{P} = \tau(P)$ that is compatible with $\widehat{J}$.

Memory Consistency: Let $\widehat{O}$ be all the operations of a computation $\widehat{C}$ of $(\tau(P),\widehat{J})$. Define the following three additional partial orders on $\widehat{O}$.

buffer-lists: $(\widehat{o_1},\widehat{o_2}) \in$ *buffer-lists* if and only if $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ and $\widehat{o_1},\widehat{o_2} \in \widehat{O}|_{\text{append} \cup \text{last}}$.

blocking-loads: $(\widehat{o_1},\widehat{o_2}) \in$ *blocking-loads* if and only if $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ and $\widehat{o_1},\widehat{o_2} \in \widehat{O}|_{\text{append} \cup \text{last} \cup \text{mem-read}}$.

FIFO-move: $(\widehat{o_1},\widehat{o_2}) \in$ *FIFO-move* if and only if $\widehat{o_1},\widehat{o_2} \in \widehat{O}|_{\text{move}}$.

Informally, buffer-lists order ensures that each read or write operation appears to be invoked in program order (as did buffer-sets order for the basic writer buffer system). Blocking-loads order ensures that when a last returns $\perp$ and hence main-memory is consulted for a value, the invoking processor waits for the mem-read to return, before initiating either another read (beginning with last) or another write (beginning with an append). The FIFO-move order ensures that pending writes from a buffer are performed in FIFO order. The buffer-lists and blocking-loads orders when added to the FIFO-per-location-memory, and matching-ops orders of the basic write-buffer system, capture the informal description of the behaviour of the partial store order write-buffer machine.

$\widehat{PSO}$ **consistency:** The computation $\widehat{C}$ satisfies $\widehat{PSO}$ *consistency* (abbreviated $\widehat{PSO}$) if there exists a valid total order of the operations $\widehat{O}$ of $\widehat{C}$ that preserves matching-ops order, buffer-lists order, FIFO-per-location-memory order and blocking-loads order.

The stronger requirement of FIFO-move, together with the orders required for $\widehat{PSO}$ consistency captures the informal description of the behaviour of the total store order write-buffer machine.

$\widehat{TSO}$ **consistency:** The computation $\widehat{C}$ satisfies $\widehat{TSO}$ *consistency* (abbreviated $\widehat{TSO}$) if there exists a valid total order of the operations $\widehat{O}$ of $\widehat{C}$ that preserves matching-ops order, buffer-lists order, FIFO-per-location-memory order, blocking-loads order and FIFO-move order.

The partial orders that define $\widehat{PSO}$ and $\widehat{TSO}$ consistency relate directly to the operational behaviour of the corresponding write-buffer machine. However, the combinations of partial orders contain redundancies. They can be simplified as follows. Define the partial order:

FIFO-per-location-move: $(\widehat{o_1}, \widehat{o_2}) \in$ *FIFO-per-location-move* if and only if $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ and $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\text{move}}|_{\widehat{x}}$, for some variable $\widehat{x} \in \widehat{J}$.

Claim 3.1. *Computation $\widehat{C}$ satisfies $\widehat{PSO}$ consistency if and only if there exists a valid total order of the operations $\widehat{O}$ of $\widehat{C}$ that preserves matching-ops order, FIFO-per-location-move order and blocking-loads order.*

Proof: For any computation that satisfies the definition of $\widehat{PSO}$ consistency there is a valid sequence of all its operations that preserves matching-ops order, buffer-lists order, FIFO-per-location-memory order and blocking-loads order. This same sequence trivially preserves the ones required for the claim.

For the other direction, let $\widehat{pso}$ be a valid sequence of all the operations $\widehat{O}$ in $\widehat{C}$, where this sequence preserves matching-ops order, FIFO-per-location-move order and blocking-loads order. Then $\widehat{pso}$ clearly preserves buffer-lists order because it is a subset of blocking-loads order. It remains to show that $\widehat{pso}$ also preserves FIFO-per-location-memory order. Let $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ and $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\text{mem-read} \cup \text{move}}|_{\widehat{x}}$, for some variable $\widehat{x}$.

- If $\widehat{o_1}$ and $\widehat{o_2}$ are both move operations then $\widehat{o_1}$ precedes $\widehat{o_2}$ in $\widehat{pso}$ because it preserves FIFO-per-location-move.
- If $\widehat{o_1}$ and $\widehat{o_2}$ are both mem-read operations then $\widehat{o_1}$ precedes $\widehat{o_2}$ in $\widehat{pso}$ because it preserves blocking-loads.
- If $\widehat{o_1}$ is a mem-read operation and $\widehat{o_2}$ is a move operation then by blocking-loads order, $\widehat{o_1}$ precedes in $\widehat{pso}$ the append that matches $\widehat{o_2}$ and by matching-ops order, this append precedes $\widehat{o_2}$.
- If $\widehat{o_1}$ is a move operation and $\widehat{o_2}$ is a mem-read operation then $\widehat{o_2}$ must have a matching last that returned $\perp$ and which precedes $\widehat{o_2}$ in $\widehat{pso}$, by matching-ops order. Operation $\widehat{o_1}$ must precede this last since otherwise the last must intervene in $\widehat{pso}$ between this move and its matching append, by the blocking-loads order. However, this requires the last to return a non-$\perp$ value for $\widehat{pso}$ to be valid because the append and last are applied to the same object. Hence, $\widehat{o_1}$ precedes this last in $\widehat{pso}$ and consequently it precedes $\widehat{o_2}$ in $\widehat{pso}$.

■

Claim 3.2. *Computation $\widehat{C}$ satisfies $\widehat{TSO}$ consistency if and only if there exists a valid total order of all the operations $\widehat{O}$ of $\widehat{C}$ that preserves matching-ops order, FIFO-move order and blocking-loads order.*

Proof: By the definitions of $\widehat{PSO}$ and $\widehat{TSO}$ consistencies, a computation satisfies $\widehat{TSO}$ consistency if and only if there is a sequence of all its operations that satisfies the requirements of $\widehat{PSO}$ consistency and preserves FIFO-move order. Thus, by Claim 3.1, a computation $\widehat{C}$ satisfies $\widehat{TSO}$ consistency if and only if there exists a valid total order of its operations that preserves matching-ops order, FIFO-per-location-move order, blocking-loads order, and FIFO-move order. Since FIFO-per-location-move order is a subset of FIFO-move order, the claim follows. ■

4 Partial and Total Store Order Consistency

One way to associate a set of computations with (P,J) is to first transform it to $(\tau(P),\widehat{J})$ (using τ as defined in Subsection 3.2) and then to consider all $\widehat{PSO}$ (or all $\widehat{TSO}$) computations that can arise. Each such computation can be interpreted as a computation of (P,J) by setting the "out" value of each read in P to the value returned by the corresponding last or mem-read. Another way is to specify a memory consistency model M and consider all computations of the system (P,J,M).

In this section, we define the two such memory consistency models PSO consistency and TSO consistency. Then we prove that the set of computations of the system (P,J,PSO) is exactly the same as the set of interpreted computations of $(\tau(P),\widehat{J},\widehat{PSO})$, and the corresponding result for TSO. Thus we establish that τ is an exact compiler from the platform (J,PSO) to the platform $(\widehat{J},\widehat{PSO})$ and from (J,TSO) to the platform $(\widehat{J},\widehat{TSO})$. Hence our "programmer level" definitions of partial and total store order computations are correct in that they exactly capture the set of possible outcomes that can arise when any multiprocess that operates on shared variables is executed on a partial or total store order write-buffer machine.

4.1 Definitions

Let O be all operations of a computation C of (P,J). $O|_{\text{write}}$ denotes the subset of O that are write operations, $O|_{\text{read}}$ is the subset of O that are read operations, and $O|_x$ is the subset of O that operate on object x. A *domestic* read operation in the computation of program p returns a value written by a write operation also by p, otherwise the read is *foreign*.

Define three partial orders on the operations O as follows.

same-object: $(o_1,o_2) \in$ *same-object* if and only if $o_1 \xrightarrow{prog} o_2$ and $o_1,o_2 \in O|_x$, for some $x \in J$.

preceding-read: $(o_1,o_2) \in$ *preceding-read* if and only if $o_1 \xrightarrow{prog} o_2$ and $o_1$is a foreign read.

following-write: $(o_1,o_2) \in$ *following-write* if and only if $o_1 \xrightarrow{prog} o_2$ and $o_2 \in O|_{\text{write}}$.

Define two weak memory consistency models using these orders.

PSO consistency: Computation C satisfies *PSO consistency* if there exists a valid total order of all its operations that preserves same-object and preceding-read orders.

TSO consistency: Computation C satisfies *TSO consistency* if there exists a valid total order of all its operations that preserves same-object, preceding-read and following-write orders.

The proofs for partial and total store order in the next two subsections are similar, so it is convenient to define some general notation and constructions that apply to either system. Denote by C_* any computation of (P,J,PSO) or (P,J,TSO). Similarly, denote by $\widehat{C_*}$ any computation of $(\tau(P),\widehat{J},\widehat{PSO})$ or $(\tau(P),\widehat{J},\widehat{TSO})$.

4.2 Transformation τ is a compiler

The definitions of PSO consistency and TSO consistency guarantee that there is a valid total order, which we denote by (O, $\xrightarrow{*so}$), on the operations of C_* that preserves some subset of the program order of P. Similarly, the definitions of $\widehat{PSO}$ and $\widehat{TSO}$ guarantee that there is a valid total order, which we denote by ($\widehat{O}$, $\xrightarrow{\widehat{*so}}$), on the operations of $\widehat{C_*}$ that preserves some subset of the program order of $\tau(P)$.

Construction one

This procedure constructs a computation, C, of (P,J) and a sequence, S, of all operations of C from a valid total order $(\widehat{O}, \widehat{*so}\rightarrow)$ of all the operations of a computation $\widehat{C_*}$ of $(\tau(P),\widehat{J},\widehat{PSO})$ or of $(\tau(P),\widehat{J},\widehat{TSO})$.

Consider the unique sequence $\widehat{*so}$ that agrees with some total order $(\widehat{O}, \widehat{*so}\rightarrow)$. Because $\widehat{*so}$ is valid, any $\text{last}_{\widehat{p_i}}(\widehat{x_i},v)$ in $\widehat{*so}$ where $v \neq \perp$ is causally related to the most recent preceding $\text{append}_{\widehat{p_i}}(\widehat{x_i},v)$ in $\widehat{*so}$. Also, each $\text{last}_{\widehat{p_i}}(\widehat{x_i},v)$ where $v \neq \perp$ occurs between its causally related $\text{append}_{\widehat{p_i}}(\widehat{x_i},v)$ and the $\text{move}_{\widehat{p_i}}(\{\widehat{x_i},\widehat{x}\},v)$ that matches this append.

Create a sequence S from $\widehat{*so}$ as follows:

reordering step Move each $\text{last}_{\widehat{p_i}}(\widehat{x_i},v)$ where $v \neq \perp$ to immediately follow the $\text{move}_{\widehat{p_i}}(\{\widehat{x_i},\widehat{x}\},v)$ that matches its causally related $\text{append}_{\widehat{p_i}}(\widehat{x_i},v)$. If multiple $\text{last}_{\widehat{p_i}}(\widehat{x_i},v)$'s with $v \neq \perp$ come between one append and its matching move, all are moved, maintaining their original order. Call this sequence *Adjust($\widehat{*so}$)*.

conversion step Delete from Adjust($\widehat{*so}$) each $\text{last}(\cdot,\perp)$ and $\text{append}(\cdot,\cdot)$ and replace each $\text{mem-read}_{\widehat{p_i}}(\widehat{x},v)$ with $\text{read}_{p_i}(x,v)$, each $\text{last}_{\widehat{p_i}}(\widehat{x_i},v)$ where $v \neq \perp$ with $\text{read}_{p_i}(x,v)$ and each $\text{move}_{\widehat{p_i}}(\{\widehat{x_i},\widehat{x}\},v)$ with $\text{write}_{p_i}(x,v)$.

Computation C of (P,J) is constructed by applying to computation $\widehat{C_*}$ the same deletions and replacements of operations as is done in the *conversion step* (above). Say that each write (respectively, read) in S *relates* to the move (respectively, mem-read or last) that it replaced in the conversion step.

For any sequence L, the notation $o_1 \xrightarrow{L} o_2$ is extended to sets of operations in the natural way. That is, for any sets of operations O_1 and O_2, $O_1 \xrightarrow{L} O_2$ means that for any $o_1 \in O_1$ and any $o_2 \in O_2$, $o_1 \xrightarrow{L} o_2$. Also, $L|_x$ denotes the subsequence of operations in L that are applied to object x. The following three claims summarize useful properties of Construction one.

Claim 4.1. *The sequence S is a valid total order of all operations in C.*

Proof: Sequence $\widehat{*so}$ is valid. Also, the write operations in S and the read operations in S that relate to mem-read operations are in the same order as the related move and mem-read operations in $\widehat{*so}$. Therefore, each read in S that relates to a mem-read returns the value of the most recent preceding write to the same variable. In the reordering step, each last that returns a non-$\perp$ value is moved so that the value it returns is the value written by the most recent preceding move to the same list object. Therefore, each read in S that relates to a last in $\widehat{*so}$ also returns the value of the most recent preceding write to the same variable. Since S contains exactly the operations in C, it is a valid total order of the operations in C. ■

Claim 4.2. *If $o_1 \xrightarrow{prog} o_2$ and o_1 is a foreign* read *in C, then $o_1 \xrightarrow{S} o_2$.*

Proof: Operation o_1 is a foreign read so it relates to a mem-read, say $\widehat{r}$, in $\widehat{C_*}$. Operation o_2 relates to either a last, mem-read or move, say $\widehat{o_2}$, in $\widehat{C_*}$. Since $o_1 \xrightarrow{prog} o_2$ in C, $\tau(o_1) \xrightarrow{\widehat{prog}} \tau(o_2)$ in $\widehat{C_*}$. So in particular, $\widehat{r} \xrightarrow{\widehat{prog}} \widehat{o_2}$. Therefore $\widehat{r} \xrightarrow{\widehat{*so}} \widehat{o_2}$ if $\widehat{o_2}$ is mem-read or last, because $\widehat{*so}$ preserves blocking-loads. Otherwise, $\widehat{o_2}$ is a move and the blocking-loads order requires $\widehat{r}$ to precede the append that matches $\widehat{o_2}$. This, with the matching-ops order, implies that $\widehat{r} \xrightarrow{\widehat{*so}} \widehat{o_2}$. The construction of S from $\widehat{*so}$ ensures that the position of o_1 in S is the same as the position of $\widehat{r}$ in $\widehat{*so}$ and the position of o_2 in S is either the same as or later than the position of $\widehat{o_2}$ in $\widehat{*so}$. It follows that $o_1 \xrightarrow{S} o_2$. ■

Claim 4.3. *If* $o_1 \xrightarrow{prog} o_2$ *and* $o_1, o_2 \in O|_x$ *for some* $x \in J$*, then* $o_1 \xrightarrow{S} o_2$.

Proof: Since $o_1 \xrightarrow{prog} o_2$ in C, $\tau(o_1) \xrightarrow{\widehat{prog}} \tau(o_2)$ in $\widehat{C_*}$. So in particular, $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ in $\widehat{C_*}$, where $\widehat{o_1}$ (respectively $\widehat{o_2}$) relates to o_1 (respectively o_2).

If o_1 and o_2 are both write operations, then $\widehat{o_1}$ and $\widehat{o_2}$ are both move operations. So, $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{o_2}$ because $\widehat{*so}$ preserves (at least) FIFO-per-location-memory order. This order is maintained by the related write operations in S implying $o_1 \xrightarrow{S} o_2$.

If o_1 is a write operation and o_2 is a read operation, then $\widehat{o_1}$ is a move operation and $\widehat{o_2}$ is either a mem-read or a last operation. If $\widehat{o_2}$ is a mem-read operation, $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{o_2}$ since $\widehat{*so}$ maintains (at least) FIFO-per-location-memory order. This order is maintained in Adjust($\widehat{*so}$) and thus by the related operations o_1 and o_2 in S, implying $o_1 \xrightarrow{S} o_2$. Otherwise, $\widehat{o_2}$ is a last operation, say $\widehat{o_2} = \text{last}_{\widehat{p_i}}(\widehat{x_i}, v)$ where $v \neq \perp$. By the validity of $\widehat{*so}$, $\text{last}_{\widehat{p_i}}(\widehat{x_i}, v)$ follows its causally related $\text{append}_{\widehat{p_i}}(\widehat{x_i}, v)$ and precedes the matching $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$. Let $\widehat{m}$ refer to $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ and let $\widehat{a}$ be its matching $\text{append}_{\widehat{p_i}}(\widehat{x_i}, v)$. The construction of Adjust($\widehat{*so}$) moved $\text{last}_{\widehat{p_i}}(\widehat{x_i}, v)$ to follow $\widehat{m}$. If $\widehat{o_1}$ is $\widehat{m}$, then in Adjust($\widehat{*so}$), $\widehat{o_2}$ was moved to immediately follow $\widehat{o_1}$, implying $o_1 \xrightarrow{S} o_2$. Now consider the case when $\widehat{o_1}$ is not $\widehat{m}$. Let $\widehat{a_1}$ be the append that matches $\widehat{o_1}$. By matching-ops order $\widehat{a_1} \xrightarrow{\widehat{*so}} \widehat{o_1}$. Then $\widehat{a_1} \xrightarrow{\widehat{*so}} \widehat{a}$ since otherwise there is a more recent append than $\widehat{a}$ (namely, $\widehat{a_1}$) that precedes $\widehat{o_2}$, which is not valid. Therefore, $\widehat{a_1} \xrightarrow{\widehat{prog}} \widehat{a}$, and consequently, $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{m}$. Since $\widehat{*so}$ preserves FIFO-per-location-memory, $\widehat{o_1}$ must precede $\widehat{m}$ in $\widehat{*so}$. In Adjust($\widehat{*so}$), $\widehat{o_2}$ was moved to immediately follow $\widehat{m}$ and, therefore, it follows $\widehat{o_1}$. Thus, $o_1 \xrightarrow{S} o_2$.

If o_1 is a read operation, then $\widehat{o_1}$ is either a mem-read operation or a last operation that returns a value other than $\perp$. In either case, $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{o_2}$ because $\widehat{*so}$ preserves blocking-loads and matching-ops orders. If $\widehat{o_1}$ is a mem-read operation then it is not moved in Adjust($\widehat{*so}$) so $o_1 \xrightarrow{S} o_2$. Otherwise, $\widehat{o_1}$ is a last operation, say $\widehat{o_1}$ is $\text{last}_{\widehat{p_i}}(\widehat{x_i}, v)$. This last is moved in Adjust($\widehat{*so}$) to immediately follow the $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ that matches the preceding and casually related $\text{append}_{\widehat{p_i}}(\widehat{x_i}, v)$. The subcases depend on what $\widehat{o_2}$ is.

If $\widehat{o_2}$ is a mem-read, its matching last must have returned $\perp$ which, by the validity of $\widehat{*so}$, implies that $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ precedes $\widehat{o_2}$ in $\widehat{*so}$. If $\widehat{o_2}$ is a move and $\widehat{a_2}$ is its matching append, then $\text{append}_{\widehat{p_i}}(\widehat{x_i}, v) \xrightarrow{\widehat{prog}} \widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{a_2}$. Hence, $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v) \xrightarrow{\widehat{prog}} \widehat{o_2}$ and consequently (again) $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ precedes $\widehat{o_2}$ in $\widehat{*so}$ because it preserves FIFO-per-location-memory. In both of these subcases $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ precedes $\widehat{o_2}$ in $\widehat{*so}$ implying $\widehat{o_1}$ precedes $\widehat{o_2}$ in Adjust($\widehat{*so}$) and hence $o_1 \xrightarrow{S} o_2$. Finally, if $\widehat{o_2}$ is a last, $\widehat{o_2}$ will also be moved to follow the move, say $\widehat{m_2}$, that matches it causally related append. If $\widehat{m_2}$ is $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ both $\widehat{o_1}$ and $\widehat{o_2}$ are moved in Adjust($\widehat{*so}$) to follow the same move, while maintaining program order, and this order is retained by the related read operations in S. So $o_1 \xrightarrow{S} o_2$. If $\widehat{m_2}$ is not $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$, then its causally related append must follow $\text{append}_{\widehat{p_i}}(\widehat{x_i}, v)$ in program order implying $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v) \xrightarrow{\widehat{prog}} \widehat{m_2}$. Hence, $\text{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v) \xrightarrow{\widehat{*so}} \widehat{m_2}$ implying $\widehat{o_1}$ precedes $\widehat{o_2}$ in Adjust$\widehat{*so}$. So in this final subcase we again have $o_1 \xrightarrow{S} o_2$ as required. ■

Lemma 4.4. *Transformation* τ *is a compiler*

1. *from* (J, PSO) *to* $(\widehat{J}, \widehat{PSO})$ *and*
2. *from* (J, TSO) *to* $(\widehat{J}, \widehat{TSO})$.

Proof: For any multiprogram P compatible with J, consider any computation $\widehat{C_*}$ of $(\tau(P),\widehat{J},\widehat{PSO})$ (respectively, $(\tau(P),\widehat{J},\widehat{TSO})$). Construct the computation C and the sequence S using construction one from the total order $\widehat{*so}$ of the operations $\widehat{O}$ in $\widehat{C_*}$. By Claim 4.1, S is a valid total order of the operations O in C. By Claim 4.3, S preserves the same-object order of O. By Claim 4.2, S preserves the preceding-read order of O. Thus, C satisfies PSO consistency and Case 1 is established.

For Case 2, it remains only to show that S preserves the following-write order. That is, we need to show that if $o_1 \xrightarrow{prog} o_2$ and $o_2 \in O|_{write}$, then $o_1 \xrightarrow{S} o_2$.

Let $\widehat{o_1}$ be the operation related (by construction one) to o_1 and $\widehat{m}$ be the move operation related to o_2. Since $o_1 \xrightarrow{prog} o_2$, $\tau(o_1) \xrightarrow{\widehat{prog}} \tau(o_2)$ in $\widehat{C_*}$ and in particular, $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{m}$.

If o_1 is a write operation then $\widehat{o_1}$ is a move operation and $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{m}$ because $\widehat{*so}$ preserves FIFO-memory order. It follows that $o_1 \xrightarrow{S} o_2$ because the order of write operations in S is the same as the order of the related move operations in $\widehat{*so}$. If o_1 is a read operation then either $\widehat{o_1}$ is a mem-read or a last operation. In either case, $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{m}$ because $\widehat{*so}$ preserves blocking-loads and matching-ops orders. If $\widehat{o_1}$ is a mem-read operation then $o_1 \xrightarrow{S} o_2$ because mem-read and move operations are never moved in Adjust($\widehat{*so}$). It remains to consider the case when $\widehat{o_1}$ is a last. In Adjust($\widehat{*so}$) $\widehat{o_1}$ is moved to immediately follow the move, say $\widehat{m_1}$ that matches its preceding and causally related append, say $\widehat{a_1}$. This $\widehat{a_1}$,$\widehat{m_1}$ pair correspond to a write operation, say w, in C. Since $\widehat{a_1} \xrightarrow{\widehat{*so}} \widehat{o_1}$, $\widehat{a_1} \xrightarrow{\widehat{prog}} \widehat{o_1}$ by blocking-loads order. Hence, $w \xrightarrow{prog} o_1$ and thus $w \xrightarrow{prog} o_2$, implying $\widehat{m_1} \xrightarrow{\widehat{prog}} \widehat{m}$. By FIFO-memory order $\widehat{m_1} \xrightarrow{\widehat{*so}} \widehat{m}$. So $\widehat{o_1}$ precedes $\widehat{m}$ in Adjust($\widehat{*so}$), implying $o_1 \xrightarrow{S} m$. ■

4.3 Transformation τ is an exact compiler

Construction two

Using the computation C_* of (P,J,PSO) or (P,J,TSO) and the valid sequence $*so$, this procedure constructs a computation $\widehat{C}$ of $(\tau(P),\widehat{J})$ and a sequence $\widehat{S}$ of all the operations, $\widehat{O}$, in $\widehat{C}$. First an intermediate computation $\widehat{D}$ and sequence $\widehat{T}$ are constructed by a straightforward conversion from C_*. Then $\widehat{D}$ and $\widehat{T}$ are adjusted to form the final computation $\widehat{C}$ and sequence $\widehat{S}$.

Construct a computation $\widehat{D}$ of $(\tau(P),\widehat{J})$ from C_* by replacing each write$_{p_i}(x,v)$ with (append$_{\widehat{p_i}}(\widehat{x_i},v)$, move$_{\widehat{p_i}}(\{\widehat{x_i},\widehat{x}\},v)$) and each read$_{p_i}(x,v)$ with (last$_{\widehat{p_i}}(\widehat{x_i},\perp)$, mem-read$_{\widehat{p_i}}(\widehat{x},v)$). An operation in C_* and the pair of operations that replaced it in $\widehat{D}$ are said to *potentially correspond*.

Sequence $\widehat{T}$ is constructed by the algorithm in Figure 4.3. The notation $\widehat{T} \leftarrow \widehat{T}$ ++ o is used to denote that operation o is appended to $\widehat{T}$. For each individual computation of p_i in C_*, maintain a pointer $\downarrow_{p_i}$ that initially points at the first operation in p_i's computation. If $\downarrow_{p_i}$ points at o, then we say $\downarrow_{p_i} = o$. Similarly, maintain the pointer $\downarrow_{*so}$ to operations in $*so$. Initially, $\downarrow_{*so}$ points at the first operation in $*so$. When there are no more operations to consider in $*so$, we say $\downarrow_{*so} = nil$. Also, advancing a pointer means the pointer is incremented to point at the next operation in the corresponding sequence. Initially, all operations in $*so$ are *unmarked*.

To create the final computation $\widehat{C}$ and sequence $\widehat{S}$, delete from both $\widehat{T}$ and $\widehat{D}$ each mem-read$(\widehat{x},v)$ operation that returns the value v of a move$(\{\widehat{x_i},\widehat{x}\},v)$ such that mem-read$(\widehat{x},v) \xrightarrow{\widehat{T}}$ move$(\{\widehat{x_i},\widehat{x}\},v)$[4]. For each

[4]Recall that all write operations (and hence all move operations) are assumed to write distinct values.

$\widehat{T} \leftarrow$ empty sequence
repeat until $\downarrow_{*so} = nil$:
  if $\downarrow_{*so} = o \in O|_{p_i}$ for some p_i and $\downarrow_{p_i} = o$ **then**
    if o is write$_{p_i}(x,v)$ **then**
      $\widehat{T} \leftarrow \widehat{T}$ ++ append$_{\widehat{p_i}}(\widehat{x_i},v)$ ++ move$_{\widehat{p_i}}(\{\widehat{x_i},\widehat{x}\},v))$
    if o is read$_{p_i}(x,v)$ **then**
      $\widehat{T} \leftarrow \widehat{T}$ ++ last$_{\widehat{p_i}}(\widehat{x_i},\perp)$ ++ mem-read$_{\widehat{p_i}}(\widehat{x},v))$
    Advance both $\downarrow_{*so}$ and $\downarrow_{p_i}$
  else if $\downarrow_{*so} = o \in O|_{p_i}$ and o is unmarked but $\downarrow_{p_i} = o' \neq o$ **then**
    if o' is write$_{p_i}(x,v)$ **then**
      $\widehat{T} \leftarrow \widehat{T}$ ++ append$_{\widehat{p_i}}(\widehat{x_i},v)$
    if o' is read$_{p_i}(x,v)$ **then**
      $\widehat{T} \leftarrow \widehat{T}$ ++ last$_{\widehat{p_i}}(\widehat{x_i},\perp)$ ++ mem-read$_{\widehat{p_i}}(\widehat{x},v)$
    Record o' as marked
    Advance $\downarrow_{p_i}$
  else ($\downarrow_{*so} = o$ and o is marked)
    if o is a write$_{p_i}(x,v)$ **then**
      $\widehat{T} \leftarrow \widehat{T}$ ++ move$_{\widehat{p_i}}(\{\widehat{x_i},\widehat{x}\},v)$
    Advance $\downarrow_{*so}$

Figure 3: Construction of $\widehat{T}$

such deleted mem-read$(\widehat{x},v)$, replace its matching last$(\widehat{x},\perp)$ by last$(\widehat{x},v)$ in both $\widehat{T}$ and $\widehat{D}$. The resulting sequence is $\widehat{S}$ and the resulting computation is $\widehat{C}$.

Claim 4.5. *Let o_1, o_2 be any* write *operations in O and let $\widehat{o_1}$, $\widehat{o_2}$ be their respectively corresponding* move *operations. If $o_1 \xrightarrow{*so} o_2$, then $\widehat{o_1} \xrightarrow{\widehat{S}} \widehat{o_2}$.*

Proof: It suffices to note that a move operation is only added to $\widehat{T}$ (and thus, $\widehat{S}$) whenever $\downarrow_{*so} = o$ and o is the corresponding write to that move operation. ■

Claim 4.6. *Let $\widehat{o_1}$, $\widehat{o_2}$ be any* append, last *or* mem-read *operations in $\widehat{O}$. If $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$, then $\widehat{o_1} \xrightarrow{\widehat{S}} \widehat{o_2}$.*

Proof: Any of these operations is only added to $\widehat{T}$ (and thus, $\widehat{S}$) whenever $\downarrow_{p_i} = o$ for each program p_i, and o corresponds to this added append, last, or mem-read operation. ■

Claim 4.7. *Let $\widehat{o_1}$ and $\widehat{o_2}$ be matching operations. If $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ then $\widehat{o_1} \xrightarrow{\widehat{S}} \widehat{o_2}$.*

Proof: If $\widehat{o_1}$ and $\widehat{o_2}$ are matching append and move operations, each append operation is added to $\widehat{T}$ either immediately preceding its corresponding move operation (i.e., when $\downarrow_{*so} = o \in O|_{p_i}$ for some p_i and $\downarrow_{p_i} = o$) or earlier (i.e., when $\downarrow_{*so} = o \in O|_{p_i}$ and o is unmarked but $\downarrow_{p_i} = o' \neq o$).

Matching last and mem-read operations are always added to $\widehat{T}$ in order of last immediately followed by mem-read. $\widehat{S}$ maintains the order of all operations that are not removed. ■

Claim 4.8. *The sequence $\widehat{S}$ is a valid total order of all operations in $\widehat{C}$.*

Proof: Exactly the same operations have been inserted into $\widehat{T}$ and into $\widehat{D}$. Also, exactly the same operations have been altered or deleted from $\widehat{T}$ and $\widehat{D}$ before renaming to $\widehat{S}$ and $\widehat{C}$. Thus, $\widehat{S}$ is a total order of the operations in $\widehat{C}$.

Sequence $*so$ is valid. Consider any read operation, $\text{read}_{p_j}(x,v)$, in $*so$. Call this read r. It returns the value of the most recent preceding write operation to the same object, $\text{write}_{p_i}(x,v)$. Call this write w. We now examine the validity of the last and mem-read operations corresponding to r that are placed onto $\widehat{S}$.

Case 1: $\downarrow_{*so}=w$ and $\downarrow_{p_j}=o_j$ where o_j is an operation satisfying $o_j \xrightarrow{prog} r$. Define I to be the interval in $*so$ between w and r and J to be the interval in the p_j's individual computation between o_j and r.

Fact 1: By the validity of $*so$, I contains no write operations to object x.

Fact 2: J contains no write operations to object x. If there were one, say $w'(x)$, then $w'(x) \xrightarrow{prog} r$ which implies $w'(x) \xrightarrow{*so} r$ by the same-object order. This implies that $w'(x) \xrightarrow{*so} w$ by fact 1. This is impossible since $\downarrow_{*so}$ advanced beyond $w'(x)$ only if $\downarrow_{p_j}$ advanced beyond $w'(x)$.

When $\downarrow_{*so}$ advances past w a corresponding move is placed onto $\widehat{T}$. The pair of operations $\text{last}(\widehat{x},\perp)$ and $\text{mem-read}(\widehat{x},v)$ that potentially correspond to r are placed onto $\widehat{T}$ when the first of $\downarrow_{*so}$ advances to r, or $\downarrow_{p_j}$ advances to r. Consider the interval K in $\widehat{T}$ between this move and this last mem-read pair.

The $\text{mem-read}(\widehat{x},v)$ is valid as long as K contains no move operation to x. Since move operations are only placed onto $\widehat{T}$ when $\downarrow_{*so}$ advances past the corresponding write, Fact 1 implies there is no such move.

The $\text{last}(\widehat{x},\perp)$ is valid as long as in $\widehat{T}$ there is no preceding append by p_j to $\widehat{x}$ without its matching move. Since append operations are only placed on $\widehat{T}$ when $\downarrow_{p_j}$ advances past a write and Fact 2 ensures a write does not exists, there is no such append in K. Furthermore, there is no such append preceding K since any write to x that preceded o_j in program order preceded w is $*so$, ensuring the matching move is on $\widehat{T}$.

Case 2: $\downarrow_{*so}=w$ and $\downarrow_{p_j}=o_j$ where o_j is an operation satisfying $r \xrightarrow{prog} o_j$. This occured because there is an operation o_2 by p_j such that $o_2 \xrightarrow{*so} w$ but $r \xrightarrow{prog} o_2 \xrightarrow{prog} o_j$. When $\downarrow_{*so}$ advanced to o_2 the pointer $\downarrow_{p_j}$ advanced past r to o_2. When $\downarrow_{p_j}$ advanced past r, it was marked and the last and mem-read operations that potentially correspond to r were placed on $\widehat{T}$.

Notice $r \xrightarrow{prog} o_2$ and $o_2 \xrightarrow{*so} r$. Thus, by preceding-read order, r cannot be a foreign read and hence the write w and the read r are both by p_j. Also, $w \xrightarrow{*so} r$ implies by same-object order that $w \xrightarrow{prog} r$. Summarizing, $o_2 \xrightarrow{*so} w \xrightarrow{*so} r$ and $w \xrightarrow{prog} r \xrightarrow{prog} o_2$.

The algorithm placed the append that corresponds to w onto $\widehat{T}$ when $\downarrow_{p_j}=w$. It placed the last and mem-read pair that potentially correspond to r onto $\widehat{T}$ when $\downarrow_{p_j}$ advanced to r. Both of these occurred before $\downarrow_{*so}$ advanced to o_j and hence before $\downarrow_{*so}$ advanced past w which places the move that corresponds to w onto $\widehat{T}$. Thus, the last and mem-read for r are between the append and move for w. Hence the final adjustment that transforms $\widehat{T}$ to $\widehat{S}$ replaces this last and mem-read pair with a last that returns the value of the append that corresponds to w, which is valid.

■

Theorem 4.9. *Transformation* τ *is an exact compiler*

1. *from* (J,PSO) *to* $(\widehat{J},\widehat{PSO})$ *and*

2. *from* (J,TSO) *to* $(\widehat{J},\widehat{TSO})$.

Proof: Since Lemma 4.4 establishes that τ is a compiler it remains only to show that for any computation, C of (P,J,PSO) (respectively, (P,J,TSO)) there is a computation $\widehat{C}$ of $(\tau(P),\widehat{J},\widehat{PSO})$ (respectively, $(\tau(P),\widehat{J},\widehat{TSO})$) whose interpretation is C.

Construct the computation $\widehat{C}$ and the sequence $\widehat{S}$ using construction two from the total order $*so$ of the operations in C. By Claim 4.8, $\widehat{S}$ is a valid total order of the operations in $\widehat{C}$. By Claims 3.1 and 3.2 it suffices to prove that $\widehat{S}$ preserves matching-ops order, FIFO-per-location-move order and blocking-loads order (for PSO) and matching-ops order, FIFO-move order and blocking-loads order (for TSO). Claim 4.6 ensures blocking-loads order and Claim 4.7 ensures matching-ops order.

By Claim 4.5, the move operations in $\widehat{S}$ are in the same order as their corresponding write operations in $*so$. If C satisfies PSO, then $*so$ preserves same-object order and, therefore, $\widehat{S}$ maintains FIFO-per-location-move order. Similarly, if C satisfies TSO, then $*so$ preserves following-write order and, therefore, $\widehat{S}$ maintains FIFO-move order.

■

5 SPARC Architecture Manual Specifications

It remains to show that the systems defined in Subsection 3.2 are equivalent to the specifications in the SPARC architecture manuals [SPA92, WG94]. The SPARC version 8 manual defines total store order and partial store order [SPA92]. The version 9 manual [WG94] redefines these two models and introduces a third model, the relaxed memory order. The manuals use both operational and axiomatic descriptions. In either case, it is difficult to interpret these models as successive weakenings of the intuitive sequential consistency. It is even more difficult to design algorithms for multiprocessor systems using these descriptions.

This section defines partial store order and total store order architecture systems using a validity condition and a program order definition that mimic those used in the version 9 architecture manual [WG94]. Then, we argue that these are equivalent to the models in the manual by proving that the partial and total store order write-buffer systems of Subsection 3.2 exactly implement the partial and total store order architecture systems (respectively). Throughout, we restrict our attention to read and write operations even though the manual defines a series of memory barrier and atomic read-modify-write operations. These can be easily added [Kaw00], but are outside the scope of this paper.

5.1 Systems based on SPARC manual specifications

SPARC *program order*, denoted by $<p$, is defined as "$X_n <p\ Y_n$ is true if and only if the memory transaction X_n is caused by an instruction that is executed before the instruction that caused memory transaction Y_n", where X_n and Y_n are executed in the same processor n (section D.3.2 of [WG94]).

The validity condition for SPARC read/write variables is more complex than the one defined in Section 2 of this paper. "The value of a load [$\cdots$] is the value of the most recent store that was performed with respect to memory order or the value of the most recent initiated store by the same processor" (section D.4.5 of [WG94].) This definition is circular. Programs are defined in terms of the objects they operate on, and the validity of these objects is defined in terms of the programs. In order to remain close to the manual specifications, we begin our definition with programs, even though there is a forward reference to objects.

The systems (P_a, J_a, TSO_a) and (P_a, J_a, PSO_a) that correspond to the manual specifications are defined using our notion of programs, objects, and memory consistency.

Programs: $P_a = \{p_1 \ldots p_n\}$ is a collection of individual programs, each containing operation invocations on SPARC variables. The order of operations in each individual program is denoted $<_p$.

Objects: A SPARC *variable* x is the set of sequences over operations of the type $\text{write}_{p_i}(x, \cdot)$ and $\text{read}_{p_j}(x, \cdot)$, $p_i, p_j \in P_a$, that satisfy the validity condition:

> the output value returned by each read_{p_i} is the input value of the latest write_{p_j} in the sequence such that:
>
> - write_{p_j} precedes read_{p_i}, or
> - write_{p_j} follows read_{p_i}, but $i = j$ and $\text{write}_{p_j} <_p \text{read}_{p_i}$.
>
> The output value is $\perp$ if no such write exists.

J_a is a set of SPARC variables.

Memory Consistency: Let O_a be all operations of a computation C_a of (P_a, J_a). Define partial orders on the operations O_a as follows:

SPARC-**same-object:** $(o_1, o_2) \in$ SPARC-same-object if and only if $o_1 <_p o_2$ and $o_1 \in O_a|_x$ and $o_2 \in O_a|_{\text{write}}|_x$.

SPARC-**read:** $(o_1, o_2) \in$ SPARC-read if and only if $o_1 <_p o_2$ and $o_1 \in O_a|_{\text{read}}$.

SPARC-**write:** $(o_1, o_2) \in$ SPARC-write if and only if $o_1 <_p o_2$ and $o_2 \in O_a|_{\text{write}}$.

Define weak memory consistency models using these orders:

PSO_a **consistency:** Computation C_a satisfies PSO_a consistency if there exists a valid total order $(O_a, \xrightarrow{pso_a})$ that preserves SPARC-same-object and SPARC-read orders.

TSO_a **consistency:** Computation C_a satisfies TSO_a consistency if there exists a valid total order $(O_a, \xrightarrow{tso_a})$ that preserves SPARC-same-object, SPARC-read and SPARC-write orders.

Because SPARC-same-object order is a subset of SPARC-write order we redefine TSO_a consistency more concisely as follows:

TSO_a **consistency:** Computation C_a satisfies TSO_a consistency if there exists a valid total order $(O_a, \xrightarrow{tso_a})$ that preserves SPARC-read and SPARC-write orders.

5.2 Equivalence with SPARC manual specifications

To see that (P_a, J_a, PSO_a) and (P_a, J_a, TSO_a) are equivalent to the systems defined in the SPARC Architecture Manual version 9 [WG94] one must carefully examine Appendix D: Formal Specifications of the Memory Model. The following discussion focuses on the memory consistency model and the validity condition of the SPARC variables of the manual.

Programs: The manual (Section D.1) defines the system to be a collection of processors, $P_0, P_1, \ldots P_{n-1}$, each with its own instruction stream, sharing address space and accessing real memory and I/O locations. Our abstraction captures this as a multiprogram $P_a = \{p_1, \ldots p_n\}$.

Objects: The validity condition of SPARC variables is based directly on the manual definition of Section D.4.5.

Memory Consistency: Sections D.5 and D.6 specify the partial and total store orders that we call PSO_a and TSO_a. The descriptions, however, rely on the RMO specification of Section D.4, where the focus is on sections D.4.4 (Memory Order Constraints) and D.4.5 (Value of Memory Transactions).

Rule (1) of D.4.4 requires that dependence order be maintained in the memory order when the preceding operation has load semantics. For PSO_a and TSO_a, the implied MEMBAR `#LoadLoad|#LoadStore` after each instruction with load semantics as specified in Section D.5 subsumes the RMO Rule (1) of D.4.4. This MEMBAR ensures that instructions with load semantics are completed before any following instructions in the program with load or store semantics. All the operations we consider have either load or store semantics. Therefore the SPARC-read order captures the constraints of the implied MEMBAR in D.5.

Rule (2) of D.4.4 refers to an explicit MEMBAR operation which is not included in our model.[5]

Rule (3) of D.4.4 is equivalent to the SPARC-same-object partial order of PSO_a. For TSO_a, the implied MEMBAR `#StoreStore` (Section D.6) after every store instruction subsumes Rule(3) of RMO. The SPARC-write order of TSO_a is equivalent for this implied MEMBAR.

Therefore, the systems (P_a, J_a, TSO_a) and (P_a, J_a, PSO_a) are faithful to the descriptions of total and partial store orders in the SPARC manuals. Next, we prove that the transformation τ of Subsection 3.2 is an exact compiler from the platform (J_a, PSO_a) to the platform $(\widehat{J}, \widehat{PSO})$ and from (J_a, TSO_a) to the platform $(\widehat{J}, \widehat{TSO})$.

5.3 Equivalence with the systems of Section 3

The ideas in the proof of the following theorem are almost identical to those in Section 4. Hence, the proof has been relegated to Appendix A.

Theorem 5.1. *Transformation τ is an exact compiler*

1. *from (J_a, PSO_a) to $(\widehat{J}, \widehat{PSO})$ and*
2. *from (J_a, TSO_a) to $(\widehat{J}, \widehat{TSO})$.*

Section 4 showed that the *PSO* and *TSO* memory consistency models capture exactly the computations that can arise on a machine with write-buffers that operates with partial store order or total store order semantics, respectively. Similarly, Section 5 shows that this same write-buffer machine captures the computations specified as PSO_a and TSO_a in the SPARC manual. Hence, our non-operational description of *PSO* and *TSO* are equivalent to the SPARC manual architectural descriptions.

6 Other Write-Buffer Architectures

The SPARC Version 9 architecture includes another memory consistency model called the relaxed memory order [WG94], which is a relaxation of *PSO*. An obvious question is whether this relaxed memory order model is also equivalent to some natural operation of a write-buffer machine. In this section, we investigate this question and extend our observations to the weak memory consistency model coherence and to the memory consistency of the Alpha multiprocessor [Cor98] as formalized by Attiya and Friedman [AF94]. We will argue that none of these memory consistency models have an exact implementation on any natural multiprocessor with write-buffers. The key is a coherent computation (defined below) that cannot occur on even a basic write-buffer machine.

[5] The MEMBAR and other similar instructions can be easily incorporated into our framework [Kaw00].

6.1 Coherence, Alpha, and RMO consistency

Coherence is a weak memory consistency condition that is sometimes assumed to be a minimum consistency requirement for any reasonable multiprocessor. Recall that J is a set of variables and P is any multiprogram compatible with J, and O is all the operations of a computation C of (P,J).

Coherence: Computation C satisfies *coherence* if for each object x, there is a valid total order of all the operations in $O|_x$ that preserves the same-object order.

According to the Alpha architecture specifications [Cor98], the memory consistency of the Alpha multiprocessor is at least as strong as coherence. The specifications state that "All processors must provide a coherent view of memory", but indicate that "Write buffers may be used to delay and aggregate writes." (pages 5-4 and 5-5 [Cor98]) From the Alpha definition by Attiya and Friedman [AF94], it appears that the Alpha multiprocessor becomes exactly coherence when no control and data dependences are present.

The relaxed memory order consistency of the SPARC machine as described in the SPARC architecture manual [WG94] allows some computations that are not coherent. The third "memory order constraint" (page 260 of [WG94]), however, requires that if two operations are on the same object, these operations must maintain their program order in the "memory order" provided at least one of them is a write. Relaxed memory order does not require program order to be maintained between two operations that are on different objects, as long as there is no "dependence" relation (page 258 of [WG94]) between them. This weakening can be expressed as a partial order:

weak same object order: $(o_1,o_2) \in$ *weak same object* if and only if $o_1 \xrightarrow{prog} o_2$ and $o_1,o_2 \in O|_x$ and $(o_1 \in O|_{\text{write}}$ or $o_2 \in O|_{\text{write}})$ for some $x \in J$.

The SPARC relaxed memory order (called *RMO*) can be defined as:

RMO consistency: Computation C satisfies *RMO consistency* if there is a valid total order of all its operations that preserves weak same object order.

6.2 Coherence, Alpha and RMO using write-buffers

Consider the following two program multiprocess $(\{p,q\},\{x,y\})$ where x and y are variables.

$$\begin{cases} p : \text{read}(x)\ \text{write}(x,1)\ \text{read}(x)\ \text{write}(y,6)\ \text{write}(y,4) \\ q : \text{read}(y)\ \text{write}(y,3)\ \text{read}(y)\ \text{write}(x,5)\ \text{write}(x,2) \end{cases}$$

The following is a computation of $(\{p,q\},\{x,y\})$.

Computation 1. $\begin{cases} p : \text{read}(x,5)\ \text{write}(x,1)\ \text{read}(x,2)\ \text{write}(y,6)\ \text{write}(y,4) \\ q : \text{read}(y,6)\ \text{write}(y,3)\ \text{read}(y,4)\ \text{write}(x,5)\ \text{write}(x,2) \end{cases}$

Observe that there is only one valid total order for variable x that maintains program order.

$$S_x = \text{write}_q(x,5)\ \text{read}_p(x,5)\ \text{write}_p(x,1)\ \text{write}_q(x,2)\ \text{read}_p(x,2).$$

Similarly, there is only one valid total order for variable y that maintains program order:

$$S_y = \text{write}_p(y,6)\ \text{read}_q(y,6)\ \text{write}_q(y,3)\ \text{write}_p(y,4)\ \text{read}_q(y,4).$$

These valid total orders confirm that Computation 1 is coherent. Notice that all read operations in computation 1 are foreign read operations. Also observe that Computation 1 satisfies neither *PSO* nor *TSO* consistency because there is no valid total order that preserves preceding-read order.

Now consider the basic write-buffer system $(\widehat{P},\widehat{K},\widehat{WB}) = (\{\gamma(p),\gamma(q)\},\{\widehat{x},\widehat{y},\widehat{x_p},\widehat{y_p},\widehat{x_q},\widehat{y_q}\},\widehat{WB})$ associated with $(\{p,q\},\{x,y\})$ where $\widehat{x}$ and $\widehat{y}$ are variables and $\widehat{x_p}$, $\widehat{y_p}$, $\widehat{x_q}$ and $\widehat{y_q}$ are set objects.

Claim 6.1. *There is no computation of $(\widehat{P},\widehat{K},\widehat{WB})$ whose interpretation is Computation 1.*

Proof: Suppose there is a computation $\widehat{C}$ of $(\widehat{P},\widehat{K},\widehat{WB})$ whose interpretation is Computation 1. Because each read in Computation 1 is a foreign read, each $\text{read}_u(z)$ for $z \in \{x,y\}$ and $u \in \{p,q\}$ is transformed to the sequence of operations: $\text{member}_{\widehat{u}}(\widehat{z_u},\perp)$, $\text{mem-read}_{\widehat{u}}(\widehat{z},v)$.

Computation $\widehat{C}$ is a basic write-buffer computation. Therefore there is a valid total order of its operations that extends buffer-sets order, matching-ops order, and FIFO-per-location-memory order. These partial orders on the operations of $\widehat{C}$ are represented in Figure 4 where buffer-ops order is represented by $\dashrightarrow$, matching-ops order by $\searrow$ and FIFO-per-location-memory by $\rightsquigarrow$.

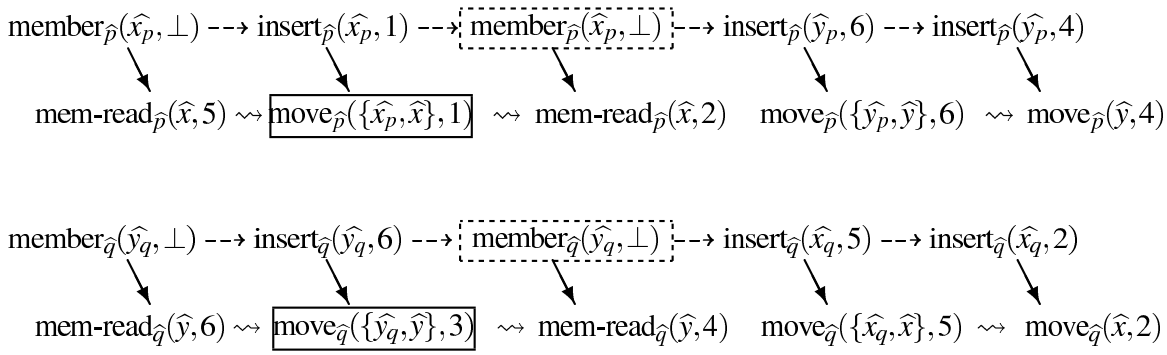

Figure 4: Orders that must be extended in $\widehat{WB}$

To ensure the validity of the two member operations in dashed boxes, each must be preceded by the move operation by the same process in the solid box since otherwise, member cannot return $\perp$. In addition, to ensure validity of the $\text{mem-read}_{\widehat{p}}(\widehat{x},5)$ and $\text{mem-read}_{\widehat{q}}(\widehat{y},6)$ operations, each must be preceded by the opposite processor's causally related move operation $\text{move}_{\widehat{q}}(\{\widehat{x_q},\widehat{x}\},5)$ and $\text{move}_{\widehat{p}}(\{\widehat{x_p},\widehat{y}\},6)$ respectively. Adding these four validity arrows creates a cycle: $\text{move}_{\widehat{p}}(\{\widehat{y_p},\widehat{y}\},6)$, $\text{mem-read}_{\widehat{q}}(\widehat{y},6)$, $\text{move}_{\widehat{q}}(\{\widehat{y_q},\widehat{y}\},3)$, $\text{member}_{\widehat{q}}(\widehat{y_q},\perp)$, $\text{insert}_{\widehat{q}}(\widehat{x_q},5)$, $\text{move}_{\widehat{q}}(\{\widehat{x_q},\widehat{x}\},5)$, $\text{mem-read}_{\widehat{p}}(\widehat{x},5)$, $\text{move}_{\widehat{p}}(\{\widehat{x_p},\widehat{x}\},1)$, $\text{member}_{\widehat{p}}(\widehat{x_p},\perp)$, $\text{insert}_{\widehat{p}}(\widehat{y_p},6)$, $\text{move}_{\widehat{p}}(\{\widehat{y_p},\widehat{y}\},6)$. Since the partial orders cannot be extended into a valid total order that preserves the write-buffer orders, computation $\widehat{C}$ does not exist. ■

Computation 1 establishes that some particular platforms cannot be exactly implemented on any machine that uses write-buffers.

Theorem 6.2. *Let M represent a coherent, RMO, or Alpha memory consistency model and J be a set of variables. Then there is no exact compiler of the platform (J,M) to any write-buffer system.*

Proof: Computation 1 was observed to be a coherent computation. Since it has no control or data dependencies, it is also an Alpha computation. Since RMO is weaker than coherence, Computation 1 is also an RMO computation. That is, Computation 1 is a computation that could occur on any coherent, RMO or Alpha platform. However, by Claim 6.1, Computation 1 is not the interpretation of any computation of any multiprogram when transformed to basic write-buffer system. Hence, Computation 1 cannot arise from any write buffer system. ■

7 Conclusions

The paper presented a framework for specifying memory consistency models and proving them correct. Distributed and parallel systems can be uniformly specified at different levels of abstraction. At any level, the system is described as a triple consisting of programs, shared objects, and a memory consistency model. The proofs establish a relationship between the components of one system at one level with the components of a corresponding system at a different level.

This paper provides simple memory consistency specifications for the SPARC version 8 architecture variants, total and partial store orders. The framework is used to prove that these non-operational specifications exactly capture the operational descriptions, and also the more complicated non-operational descriptions of the subsequent SPARC version 9 manual. These equivalences highlight some serious flaws in other specification attempts or even in the official manual specifications, such as RMO and Alpha consistency.

The paper also defines the minimum consistency guarantees that would arise from any reasonable implementation on a machine with write-buffers of a multiprogram that uses shared variables. This allows us to prove that each of several well known memory consistency models does no exactly correspond to any implementation that uses write-buffers. For instance, RMO as described in the official SPARC version 9 manual [WG94] cannot be a description of a write-buffer machine. We conjecture that RMO was intended to be such a machine, but the approach that was taken in the manual to describe it was not faithful to the machine's operation. As another example, the memory consistency of the Alpha multiprocessor [Cor98] as formalized by Attiya and Friedman [AF94] cannot be described as a basic write-buffer system. The DEC-Alpha reference manually explicitly states that write-buffers can be utilized in Alpha multiprocessors (see pages 5-4 and 5-5 of [Cor98]). However, we have shown that any such utilization of write-buffers would necessarily give a memory consistency model more constrained than that of Alpha. In particular, a memory consistency model cannot be exactly implemented as a write-buffer machine, unless it provides consistency guarantees that are stronger than coherence.

Many concerns such as asynchrony, weak consistency and faults make implementation of correct distributed systems very difficult. The task of reasoning about the correctness of these systems is somewhat eased if the parallel and distributed application developers can assume a strong consistency model such as sequential consistency or linearizability. This motivates us to hide the consistency model complexities from application developers by building compliers that transform a program that is correct under sequential consistency to an equivalent program that is also correct under weaker consistency [HK03] model. Necessary initial steps include understanding exactly what constraints are guaranteed by a given weak system, and to polish techniques that prove the correctness of these compilers, as is demonstrated in this paper.

References

[ABJ+93] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proc. 5th Int'l Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.

[ACFW98] Hagit Attiya, Soma Chaudhuri, Rob Friedman, and Jennifer Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. *SIAM Journal of Computing*, 27(1):65–89, February 1998.

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high performance multiprocessors. In *Proc. 24th Int'l Symp. on Theory of Computing*, pages 679–690, 1992.

[AF94] Hagit Attiya and Roy Friedman. Programming DEC-Alpha based multiprocessors the easy way. In *Proc. 6th Int'l Symp. on Parallel Algorithms and Architectures*, pages 157–166, 1994. Technical Report LPCR 9411, Computer Science Department, Technion.

[ANB+95] Mustaque Ahamad, Gil Neiger, James Burns, Prince Kohli, and Phillip Hutto. Causal memory: Definitions, implementations, and programming. *Distributed Computing*, 9:37–49, 1995.

[Ang89] Frank Anger. On Lamport's interprocessor communication model. *ACM Trans. on Programming Languages and Systems*, 11:404–417, July 1989.

[Cor98] Compaq Computer Corportaion. *The Alpha Architecture Handbook*. Compaq Computer Corporation, 1998. Order number: EC-QD2KC-TE.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Int'l Symp. on Computer Architecture*, pages 434–442, June 1986.

[Fri95] Roy Friedman. Implementing hybrid consistency with high-level synchronization operations. *Distributed Computing*, 9(3):119–129, December 1995.

[GM92] Phillip Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *Proc. 4th Int'l Symp. on Parallel Algorithms and Architectures*, pages 306–315, 1992.

[Goo89] James Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[GS00] Alex Gontmakher and Assaf Schuster. Java consistency: nonoperational characterizations for Java memory behavior. *ACM Trans. on Computer Systems*, 18(4):333–386, 2000.

[HK97] Lisa Higham and Jalal Kawash. Critical sections and producer/consumer queues in weak memory systems. In *Proc. 1997 IEEE Int'l Symp. on Parallel Architectures, Algorithms, and Networks*, pages 56–63, December 1997.

[HK98] Lisa Higham and Jalal 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.

[HK00] Lisa Higham and Jalal Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *Proc. of the 7th Int'l Conf. on High Performance Computing, Lecture Notes in Computer Science volume 1970*, pages 355–366, December 2000.

[HK03] Lisa Higham and Jalal Kawash. Implementing sequentially consistent programs on processor consistent platforms. Submitted, 2003.

[Hoa72] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating System Techniques*. Academic Press, 1972.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Int97] Int'l Business Machines Corporation. PowerPC microprocessor family: the programming environments for 32-bit microprocessor. http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC, 1997.

[Int02] Intel Corporation. Intel itanium architecture software developer's manual, volume 2: System architecture. http://www.intel.com/, Oct 2002.

[Kaw00] Jalal Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems.* Ph.D. dissertation, Department of Computer Science, The University of Calgary, January 2000.

[KNA93] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. In *Proc. 1993 Int'l Conf. on Parallel Processing*, August 1993.

[Lam79a] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[Lam79b] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. on Programming Languages and Systems*, 1(1):84–97, July 1979.

[Lam86a] Leslie Lamport. The mutual exclusion problem (parts I and II). *Journal of the ACM*, 33(2):313–326 and 327–348, April 1986.

[Lam86b] Leslie Lamport. On interprocess communication (parts I and II). *Distributed Computing*, 1(2):77–85 and 86–101, 1986.

[Lam97] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. on Computers*, 46(7):779–782, July 1997.

[Mis86] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. on Programming Languages and Systems*, 8(1):142–153, 1986.

[OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[PD99] Seugjoon Park and David Dill. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers*, 48(2):227–235, February 1999.

[SPA92] SPARC Int'l, Inc. *The SPARC Architecture Manual version 8.* Prentice-Hall, 1992.

[WG94] David Weaver and Tom Germond, editors. *The SPARC Architecture Manual version 9*. Prentice-Hall, 1994.

A Proof of Theorem 5.1

The proof of Theorem 5.1 is similar to the proof of Theorem 4.9. A new construction is used to show that τ is a compiler, then a previous construction, Construction two, is used to establish the compiler is exact. Due to the different validity condition in the definition of the variables, the lemmas associated with Construction two must be re-established.

A.1 Transformation τ is a compiler

Construction three

This procedure constructs a computation, C_a, of (P_a, J_a) and a sequence, S_a, of all the operations of C_a from a valid total order $(\widehat{O}, \xrightarrow{\widehat{*so}})$ of all the operations of a computation $\widehat{C_*}$ of $(\tau(P_a), \widehat{J}, \widehat{PSO})$ or of $(\tau(P_a), \widehat{J}, \widehat{TSO})$[6]. Let $\widehat{*so}$ be the unique sequence that agrees with $(\widehat{O}, \xrightarrow{\widehat{*so}})$. Create the sequence S_a from $\widehat{*so}$ and the computation C_a from $\widehat{C_*}$ as follows:

> Delete each last$(\cdot, \perp)$ and each append$(\cdot,\cdot)$ and replace each mem-read$_{\widehat{p_i}}(\widehat{x}, v)$ with read$_{p_i}(x, v)$, each last$_{\widehat{p_i}}(\widehat{x_i}, v)$ where $v \neq \perp$ with read$_{p_i}(x, v)$ and each move$_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ with write$_{p_i}(x, v)$.

Say that each write (respectively, read) in S_a *relates* to the move (respectively, mem-read or last) that it replaced in this construction. The following three claims set the stage for proving Lemma A.4, which establishes the first direction for Theorem 5.1.

Claim A.1. *The sequence S_a is a valid total order of all operations in C_a.*

Proof: Because $\widehat{*so}$ is a total order of all operations in $\widehat{C_*}$, the sequence S_a created by Construction three is a total order of the operations in C_a. It remains to show S_a is valid. The sequence $\widehat{*so}$ is valid (that is, using the validity condition of subsection 5.1.)

By Construction three and the validity of $\widehat{*so}$, any read that relates to a mem-read will return the value of the most recent write to the same object that precedes the read in S_a. However, we must show that there is no other write w' to that object by the same process such that w' precedes this read in $<_p$ of C_a and follows it in S_a. Suppose such a w' exists. Let $\widehat{mr}$ be the related operation to this read r and $\widehat{l}$ be its matching last and let $\widehat{m'}$ be the related operation to w' and $\widehat{a'}$ its matching append. Because Construction three gives $w' <_p r$, it must be the case that $\widehat{m'} \xrightarrow{\widehat{prog}} \widehat{mr}$ in $\widehat{C_*}$. The fact that $\widehat{C_*}$ is a computation of $\tau(P_a)$ gives $\widehat{a'} \xrightarrow{\widehat{prog}} \widehat{m'} \xrightarrow{\widehat{prog}} \widehat{l} \xrightarrow{\widehat{prog}} \widehat{mr}$. Since $\widehat{*so}$ maintains blocking-loads order, $\widehat{a'} \xrightarrow{\widehat{*so}} \widehat{l}$. Since $r \xrightarrow{S_a} w'$, $\widehat{l} \xrightarrow{\widehat{*so}} \widehat{m'}$. However, this requires $\widehat{l}$ to return a value other than $\perp$. This contradicts the existence of such a w'.

Now consider a read r that relates to a non-$\perp$ last operation $\widehat{l}$. It returns the value of a later write w in S_a whose related move $\widehat{m}$ matches the most recent preceding append $\widehat{a}$ in $\widehat{*so}$ by the same process to the same object. This $\widehat{a}$ is causally related to $\widehat{l}$. By the validity of $\widehat{*so}$, we have $\widehat{a} \xrightarrow{\widehat{*so}} \widehat{l} \xrightarrow{\widehat{*so}} \widehat{m}$. Since $\widehat{a} \xrightarrow{\widehat{*so}} \widehat{l}$ and $\widehat{*so}$ satisfies blocking-loads order it must be that $\widehat{a} \xrightarrow{\widehat{prog}} \widehat{l}$ in $\widehat{C_*}$. Since $\widehat{C_*}$ is a computation of $\tau(P_a)$ we must also have $\widehat{a} \xrightarrow{\widehat{prog}} \widehat{m} \xrightarrow{\widehat{prog}} \widehat{l}$. Hence, $w <_p r$ in C_a. It remains to show that there is no later write w' to the same object by

[6]Where τ is the transformation defined in subsection 3.2, used here to transform from J_a to $\widehat{J}$.

the same process as r where $w \xrightarrow{S_a} w'$ and $w' <_p r$. Let w' be any write after w in S_a. Its related move $\widehat{m'}$ must follow $\widehat{m}$ in $\widehat{*so}$. Since $\widehat{*so}$ preserves FIFO-per-location-memory order $\widehat{m} \xrightarrow{\widehat{prog}} \widehat{m'}$. By blocking-loads order the append $\widehat{a'}$ that matches $\widehat{m'}$ must follow $\widehat{a}$ in $\widehat{*so}$. But then by the validity of $\widehat{l}$ (that is, the assumption that $\widehat{a}$ was the most recent preceding append in $\widehat{*so}$), $\widehat{l} \xrightarrow{\widehat{*so}} \widehat{a'}$. Thus $r <_p w'$. Thus, w is the latest write in S_a that proceeds r in program order and therefore S_a is valid.

■

Claim A.2. *If $o_1 <_p o_2$ and o_1 is a read in C_a, then $o_1 \xrightarrow{S_a} o_2$.*

Proof: Operation o_1 relates to $\widehat{o_1}$ in $\widehat{C_*}$ and $\widehat{o_1}$ is either a mem-read or a last. Operation o_2 relates to either a last, mem-read or move, say $\widehat{o_2}$, in $\widehat{C_*}$. Since $o_1 <_p o_2$ in C_a, $\tau(o_1) \xrightarrow{\widehat{prog}} \tau(o_2)$ in $\widehat{C_*}$. So in particular, $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$. Therefore $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{o_2}$ if $\widehat{o_2}$ is mem-read or last, because $\widehat{*so}$ preserves blocking-loads order. Otherwise, $\widehat{o_2}$ is a move and the blocking-loads order requires $\widehat{o_1}$ to precede the append that matches $\widehat{o_2}$. This with the matching-ops order imply that $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{o_2}$. The order of the operations is not altered by the construction algorithm, so it follows that $o_1 \xrightarrow{S_a} o_2$. ■

Claim A.3. *If $o_1 <_p o_2$ and $o_1 \in O_a|_x$ and $o_2 \in O_a|_{write}|_x$ for some $x \in J_a$, then $o_1 \xrightarrow{S_a} o_2$.*

Proof: Since $o_1 <_p o_2$ in C_a, $\tau(o_1) \xrightarrow{\widehat{prog}} \tau(o_2)$ in $\widehat{C_*}$. So in particular, $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{o_2}$ in $\widehat{C_*}$, where $\widehat{o_1}$ (respectively $\widehat{o_2}$) relates to o_1 (respectively o_2).

Operation o_2 is related to a move operation. If o_1 is a write operation, and thus related to a move, or a read that is related to a mem-read operation, then $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{o_2}$ because $\widehat{*so}$ preserves (at least) FIFO-per-location-memory order. This order is maintained by the related write and read operations in S_a implying $o_1 \xrightarrow{S_a} o_2$.

Otherwise, o_1 is a read operation and $\widehat{o_1}$ is a last operation that returns a value other than $\perp$. The blocking-loads order requires that $\widehat{o_1}$ precedes the append that matches $\widehat{o_2}$, and the append precedes $\widehat{o_2}$ by the matching-ops order. This order is maintained by the related read and write operations in S_a implying $o_1 \xrightarrow{S_a} o_2$.

■

Lemma A.4. *Transformation τ is a compiler*

1. *from (J_a, PSO_a) to $(\widehat{J}, \widehat{PSO})$ and*
2. *from (J_a, TSO_a) to $(\widehat{J}, \widehat{TSO})$.*

Proof: For any multiprogram P_a compatible with J_a, consider any computation $\widehat{C_*}$ of $(\tau(P_a), \widehat{J}, \widehat{PSO})$ (respectively, $(\tau(P_a), \widehat{J}, \widehat{TSO})$). Construct the computation C_a and the sequence S_a using Construction three from the total order $\widehat{*so}$ of the operations in $\widehat{C_*}$. By Claim A.1, S_a is a valid total order of all operations O_a in C_a. By Claim A.3, S_a preserves the SPARC-same-object order. By Claim A.2, S_a preserves the SPARC-read order of O_a. Thus, C_a satisfies PSO_a consistency and Case 1 is established.

For Case 2, it remains only to show that S_a preserves the SPARC-write order. That is, we need to show that if $o_1 <_p o_2$ and $o_2 \in O_a|_{write}$, then $o_1 \xrightarrow{S_a} o_2$. Let $\widehat{o_1}$ be the operation related (by Construction three) to o_1 and $\widehat{m}$ be the move operation related to o_2. Since $o_1 <_p o_2$, $\tau(o_1) \xrightarrow{\widehat{prog}} \tau(o_2)$ in $\widehat{C_*}$ and in particular, $\widehat{o_1} \xrightarrow{\widehat{prog}} \widehat{m}$.

If o_1 is a write operation and $\widehat{o_1}$ is a move operation or o_1 is a read operation and $\widehat{o_1}$ is a mem-read operation then $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{m}$ because $\widehat{*so}$ preserves FIFO-memory order and it follows that $o_1 \xrightarrow{S_a} o_2$. If o_1 is a read operation and $\widehat{o_1}$ is a last operation that returns a value other than $\perp$, then $\widehat{o_1} \xrightarrow{\widehat{*so}} \widehat{m}$ because $\widehat{*so}$ preserves blocking-loads and matching-ops orders. This order is maintained by the related read and write operations in S_a implying $o_1 \xrightarrow{S_a} o_2$.

■

A.2 Transformation τ is an exact compiler

Let $(O_a, \xrightarrow{*so_a})$ be a valid total order of all the operations of a computation C_a of (P_a, J_a, PSO_a) or (P_a, J_a, TSO_a). Let $*so_a$ be the unique sequence that agrees with $(O_a, \xrightarrow{*so_a})$. Construction two of Section 4 is applied to the computation C_a of (P_a, J_a, PSO_a) or (P_a, J_a, TSO_a) and the sequence $*so_a$, yielding a computation $\widehat{C}$ of $(\tau(P_a), \widehat{J})$ and a sequence $\widehat{S}$ of all the operations, $\widehat{O}$, in $\widehat{C}$. The proofs of Claims 4.5, 4.6 and 4.7 remain true when $*so_a$ replaces $*so$ as an input sequence for Construction two. Since $*so_a$ satisfies a different validity condition from $*so$ the following claim is required to replace Claim 4.8.

Claim A.5. *The sequence $\widehat{S}$ is a valid total order of all operations in $\widehat{C}$.*

Proof: Exactly the same operations have been inserted into $\widehat{T}$ and into $\widehat{D}$. Also, exactly the same operations have been altered or deleted from $\widehat{T}$ and $\widehat{D}$ before renaming to $\widehat{S}$ and $\widehat{C}$. Thus, $\widehat{S}$ is a total order of the operations in $\widehat{C}$.

Sequence $*so_a$ is a valid sequence of operations on SPARC variables. Consider any read operation, $\text{read}_{p_j}(x,v)$, in $*so_a$. Call this read r. It returns the value of the latest in $*so_a$ of either 1) the most recent write to the same object that precedes r in p_j's program and follows r in $*so_a$, or 2) the most recent preceding write operation to the same object in $*so_a$. Call this write w. We now examine the validity of the last and mem-read operations corresponding to r that are placed onto $\widehat{S}$.

Case 1: $w <_p r$ and $r \xrightarrow{*so_a} w$. In this case w has the form $\text{write}_{p_j}(x,v)$.

> **Fact A:** For any read by p_j $\downarrow_{p_j}$ and $\downarrow_{*so_a}$ advance simultaneously. The algorithm never advances $\downarrow_{*so_a}$ past an operation earlier than $\downarrow_{p_j}$ advances past the same operation. Pointer $\downarrow_{p_j}$ could only advance beyond a read r' earlier than $\downarrow_{*so_a}$ advances beyond r' if some operation o is $o \xrightarrow{*so_a} r'$ and $r' <_p o$. But SPARC-read order ensures this does not occur.

By Fact A, $\downarrow_{*so_a}$ advances past r when $\downarrow_{p_j}$ also advances past r. When this occurs the last and mem-read that potentially correspond to r are placed onto $\widehat{T}$. Since $w <_p r$ and $r \xrightarrow{*so_a} w$ the append that corresponds to w was placed onto $\widehat{T}$ before these last and mem-read operations and the matching move after. The final adjustment that transforms $\widehat{T}$ to $\widehat{S}$ replaces this last and mem-read pair with a last that returns the value of the append that corresponds to w, which is valid.

Case 2: $w <_p r$ and $w \xrightarrow{*so_a} r$. In this case w has the form $\text{write}_{p_i}(x,v)$. There does not exist a write operation, w', such that $w' <_p r$ and $r \xrightarrow{*so_a} w'$ because that would be case 1 above. Consider when $\downarrow_{*so_a} = w$ and $\downarrow_{p_j} = o_j$ where o_j is an operation satisfying $o_j <_p r$. Define I to be the interval in $*so_a$ between w and r and J to be the interval in the p_j's individual computation between o_j and r.

> **Fact B:** By the validity of $*so_a$, I contains no write operations to object x.

Fact C: J contains no write operations to object x. If there were one, say $w'(x)$, then $w'(x)$ cannot follow r in $*so_a$ because that was case 1. $w'(x)$ is not in I by Fact B, so $w'(x) \xrightarrow{*so_a} w$. When $\downarrow_{*so_a}$ advances past $w'(x)$ it also advances $\downarrow_{p_j}$ past $w'(x)$.

When $\downarrow_{*so_a}$ advances past w a corresponding move is placed onto $\widehat{T}$. The pair of operations last$(\widehat{x}, \perp)$ and mem-read$(\widehat{x}, v)$ that potentially correspond to r are placed onto $\widehat{T}$ when the first of $\downarrow_{*so_a}$ advances to r, or $\downarrow_{p_j}$ advances to r. Consider the interval K in $\widehat{T}$ between this move and this last mem-read pair.

The mem-read$(\widehat{x}, v)$ is valid as long as K contains no move operation to x. Since move operations are only placed onto $\widehat{T}$ when $\downarrow_{*so_a}$ advances past the corresponding write, Fact B implies there is no such move.

The last$(\widehat{x}, \perp)$ is valid as long as in $\widehat{T}$ there is no preceding append by p_j to $\widehat{x}$ without its matching move. Since append operations are only placed on $\widehat{T}$ when $\downarrow_{p_j}$ advances past a write and Fact B ensures a write does not exists, there is no such append in K. Furthermore, there is no such append preceding K since any write to x that preceded o_j in program order preceded w in $*so_a$, ensuring the matching move is on $\widehat{T}$.

Thus, the mem-read and last operations that are appended in this case are valid.

■

Theorem 5.1. *Transformation* τ *is an exact compiler*

1. *from* (J_a, PSO_a) *to* $(\widehat{J}, \widehat{PSO})$ *and*
2. *from* (J_a, TSO_a) *to* $(\widehat{J}, \widehat{TSO})$.

Proof: From Lemma A.4, τ is a compiler. It remains to show that for any computation, C_a of (P_a, J_a, PSO_a) (respectively, (P_a, J_a, TSO_a)) there is a computation $\widehat{C}$ of $(\tau(P_a), \widehat{J}, \widehat{PSO})$ (respectively, $(\tau(P_a), \widehat{J}, \widehat{TSO})$) whose interpretation is C_a.

Construct the computation $\widehat{C}$ and the sequence $\widehat{S}$ using Construction two from the total order $*so_a$ of the operations in C_a. By Claim A.5, $\widehat{S}$ is a valid total order of $\widehat{C}$. By Claims 3.1 and 3.2 it suffices to prove that $\widehat{S}$ preserves matching-ops order, FIFO-per-location-move order and blocking-loads order for PSO, and additionally, FIFO-move order for TSO. Claim 4.6 ensures blocking-loads order and Claim 4.7 ensures matching-ops order.

By Claim 4.5, the move operations are in the same order as their corresponding write operations in $*so_a$. If C_a satisfies PSO_a, then $*so_a$ preserves SPARC-same-object order, which ensures that all write operations to the same object maintain program order. Therefore $\widehat{S}$ maintains FIFO-per-location-move order. Similarly, if C_a satisfies TSO_a, then $*so_a$ preserves SPARC-write order which ensures that all write operations maintains program order. Therefore $\widehat{S}$ maintains FIFO-move order.

■