Specifying Memory Consistency of Write Buffer Multiprocessors

LISA HIGHAM
The University of Calgary
LILLANNE JACKSON
The University of Calgary
and
JALAL KAWASH
American University of Sharjah

Write buffering is one of many successful mechanisms that improves the performance and scalability of multiprocessors. However, it leads to more complex memory system behaviour, which cannot be described using intuitive consistency models, such as Sequential Consistency. It is crucial to provide programmers with a specification of the exact behaviour 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 simple 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 technique is used to show that Coherence does not correspond to a write-buffer architecture. A corollary, which follows from the relationship between Coherence and Alpha, 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.

Categories and Subject Descriptors: C.0 [**General**]: Modeling of Computer Architecture; C.1.4 [**Processor Architecture**]: Parallel Architectures—*Distributed Architectures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and Concurrency*

General Terms: Theory

Additional Key Words and Phrases: Memory consistency framework, Write-buffer architectures, SPARC multiprocessors, Total store order, Partial store order, Relaxed memory order, Sequential Consistency, Alpha, Coherence

Corresponding Author: L. Higham, Department of Computer Science, The University of Calgary, 2500 University Dr. NW, Calgary, AB, Canada T2N1N4, `higham@cpsc.ucalgary.ca`.
L. Jackson is currently with the Department of Computer Science, University of Victoria, Canada.
J. Kawash is also affiliated with the Department of Computer Science, The University of Calgary, Canada.

1. INTRODUCTION

Distributed algorithm designers typically assume strong consistency models such as Sequential Consistency and Linearizability [Lamport 1979a; Herlihy and Wing 1990]. 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 behaviour of such complex memory systems resulted in incomplete or incorrect conclusions. Thus, such specifications must be shown to exactly capture the operational behaviour 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 a generalization of 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. One represents the "bare minimum" write-buffer machine capturing the very basic behaviour of write-buffers. The other two successively stronger machines capture the operational behaviour of SPARC's partial and total store order machines [SPARC Int'l, Inc. 1992; Weaver and Germond 2000]. Using the framework, three sets of increasingly stringent constraints on outcomes of program executions on these machines are specified.

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 behaviour 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 [Weaver and Germond 2000] 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 write-buffer multiprocessor. In fact, we prove that Coherence [Dubois et al. 1986; Goodman 1989] or any weaker model does not correspond to a write-buffer architecture. We also prove that Alpha consistency [Compaq Computer Corp. 1998], which is Coherence combined with some dependencies [Attiya and Friedman 1994] that arise from processors' private registers, admits computations that cannot arise on any write-buffer machine. Thus any implementation of Alpha consistency using write-buffers must produce a memory consistency model strictly stronger than that specified by the Alpha documentation [Compaq Computer Corp. 1998], 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. This applies to the SPARC architecture, with which the UltraSPARC I, II, III, and IV processors are all compliant [Sun Microsystems 2004], since it continues to be one of the commercial choices for server machines. Typically, hardware architecture manuals use natural language or axiomatic specifications to describe how the hardware operates [SPARC Int'l, Inc. 1992; Weaver and Germond 2000; Intel Corp. 2002; Int'l Business Machines Corp. 1997]. 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. The literature contains many examples of such frameworks [Hoare 1972; Owicki and Gries 1976; Misra 1986; Lamport 1979b; 1997; 1986b; 1986a; Anger 1989; Attiya et al. 1998; Attiya and Friedman 1992; Friedman 1995; Ahamad et al. 1993; Ahamad et al. 1995; Kohli et al. 1993; Gibbons and Merritt 1992; Lynch and Tuttle 1989; Lynch 1996; Herlihy and Wing 1990; Adir et al. 2003]. A comparison of our framework to the closely related ones is provided in Subsection 2.3 after ours is defined.

Capturing the consistency of a multiprocessor architecture simply and precisely at a programmer oriented level is crucial for the development of correct and efficient programs targeted to these architectures. But this has proven to be surprisingly tricky for several machines. Erroneous definitions may lead to incorrect programs. Complex definitions lead programmers to unnecessarily and aggressively use expensive synchronization primitives, which negatively impact program performance. For example, one earlier attempt to define the semantics of the SPARC memory consistency model called total store order [Kohli et al. 1993] resulted in a definition that is stronger than what the machine level architecture description in the SUN Microsystems manuals [SPARC Int'l, Inc. 1992; Weaver and Germond 2000] actually provides. 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 strong definition [Higham and Kawash 2000]. Thus using this definition, there is a solution to the mutual exclusion problem that uses only read-write variables. In earlier work [Higham and Kawash 2000; Kawash 2000], we exploit the definitions derived and proven correct in this paper to prove, to the contrary, that read-write variables are insufficient to solve the mutual exclusion problem on a SPARC total store order machine. The total and partial store order definitions derived in this paper are used in another work [Higham and Kawash 2005], to determine under what conditions wait-free producer-consumer coordination is possible or impossible in various SPARC models without resorting to expensive synchronization primitives, in spite of the impossibility of mutual exclusion.

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 also facilitates the comparison of the SPARC models to each other and to proposed consistency models including Processor Consistency [Goodman 1989], Causal Consistency [Ahamad et al. 1995], and Java consistency [Gontmakher and Schuster 2000; Higham and Kawash 1998], and aids the development of verification tools [Park and Dill 1999].

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 describe two write-buffer machines that 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. A computation is given in Section 6 that is possible on any of the memory consistency models defined by the SPARC relaxed memory order, Alpha consistency, and Coherence yet cannot arise from any write-buffer machine — even one substantially more permissive than a partial store order write buffer machine. Thus these consistency models are not equivalent to any write buffer model. Section 7 summarizes and concludes.

2. A CONSISTENCY MODELLING FRAMEWORK

2.1 Describing systems

A multiprocess system can be modelled 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. Two running examples are provided and will be used in subsequent sections of the paper.

Shared data objects: A shared data object can 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 *out*←ACT*(obj,in)* 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 out←ACT(obj,in) has two components; its *invocation* is 'ACT(obj,in)' and its *response* is 'out'. For a more concise notation, if an operation does not generate an 'out' value, it is written as ACT*(obj,in)*. If it does not require an input value, 'in', it is written as *out*←ACT*(obj)*.

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. We say that a sequence S of operations and the sequence S' of operation invocations formed by removing the response component of each operation in S are *associated* sequences.

Throughout, we assume that for each state-change operation the input parameter has a distinct value. A footnote will indicate when this assumption is being used. This is a common assumption when defining consistency models. It is not essential but removing it adds messiness. Section 7 comments further on this assumption and its implications. Also, for this paper, any non-empty input or output list contains only one item and is therefore abbreviated by omitting list delimiters. An *operation type* is just an operation that has one or more parameters chosen from some given set. The set of possible output parameters is indicated by Δ, input parameters by $\cdot$, and object parameters by $\cdot$. For example, the notation WRITE$(x,\cdot)$ denotes the set of WRITE operations on object x, or (depending on context) any element of that set; $\Delta\leftarrow$READ$(\cdot)$ denotes the set of all READ operations.

For most of this paper, we consider only two types of objects: read/write objects (which we call variables) and list objects. Section 6 introduces a set object to support our impossibility results. Lists (respectively, sets) will be used to capture the behaviour of ordered (respectively, unordered) write-buffers.

Example 1a – variable: A *variable*, x, is the set of sequences over operations of the type WRITE$(x,\cdot)$ and $\Delta\leftarrow$READ(x) 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 1b – list object: In a sequence of operations that includes types APPEND$(l,\cdot)$ and DELETE$(l,\cdot)$, an $a =$ APPEND(l,x) has a *matching delete* if there is a DELETE(l,x) that follows a in the sequence. Otherwise it is *undeleted.*[1]

A *list object*, l, is the set of sequences over operations of the type APPEND$(l,\cdot)$, DELETE$(l,\cdot)$, and $\Delta\leftarrow$LAST(l) that satisfy the validity condition:

> For any LAST operation on list object, l, it returns a value $\rho \neq\perp$ if and only if the most recent preceding undeleted APPEND operation is APPEND(l,ρ). 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

[1]This definition exploits the assumption of distinct input values.

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

variables, a READ operation returns the value of a preceding WRITE operation; and for list objects LAST returns the value of the most recent preceding undeleted APPEND operation. In each case, the output-generating operation and the unique related state-change operation are said to be *causally related*.

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 2a – variable multiprogram: Let x, y and z be variables, and let $\{p, q\}$ be the multiprogram:

$$\begin{cases} p : \text{WRITE}(x,1)\ \ \text{WRITE}(z,2)\ \ \text{READ}(x)\ \ \text{READ}(y) \\ q : \text{WRITE}(y,3)\ \ \text{READ}(y)\ \ \text{READ}(z)\ \ \text{READ}(x) \end{cases}$$

Then $(\{p, q\}, \{x, y, z\})$ is a (variable) multiprocess and $\{p, q\}$ is compatible with $\{x, y, z\}$.

Example 2b – list object multiprogram: Let l be a list object, and let $\{p, q\}$ be the multiprogram:

$$\begin{cases} p : \text{APPEND}(l,4)\ \ \text{APPEND}(l,6)\ \ \text{LAST}(l) \\ q : \text{APPEND}(l,7)\ \ \text{DELETE}(l,6)\ \ \text{DELETE}(l,4)\ \ \text{LAST}(l) \end{cases}$$

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

The preceding two program examples are particularly simple because they are only invocations on shared data objects and they lack any control structures. More elaborate programs could contain code that includes branches and loops.

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

Memory consistency models: Informally, when a multiprogram is executed, the individual programs of the processes somehow interact, and each individual program's control structures gives rise to a path through its code, producing responses to the output-generating instruction invocations on these paths. Due to control structures and non-deterministic timing and interactions, different executions of the same multiprogram can, in general, generate different individual paths, and each set of paths might generate different sets of responses depending on the rules governing the interactions. For a Sequentially Consistent [Lamport 1979a] multiprocess, any such execution can be modelled as a single valid sequence of operations that arises from some interleaving of the operation invocations generated by paths through the individual programs of the multiprocess. In weaker memory consistency models, no such straightforward interleaving is guaranteed.

To model the more general settings, we first define a computation. An *individual computation of a program* is a sequence of operations. The order of the operations in this sequence is called *program order* and must be the same as the order in which the associated operation invocations appear in some path through the individual program. A *(system) computation* of a multiprocess (P, J) is a collection of individual computations, one for each individual program, $p \in P$. For this paper it suffices to consider completed (as opposed to partial) computations of a multiprogram; that is, the individual computations of a multiprogram must be associated with the sequences formed from all the operation invocations in the paths executed by the multiprogram. Therefore, when the individual programs of a multiprogram P are each straight line programs as in our examples above, a computation of P is just like P except each operation invocation is completed to an operation; that is, all "out" values are filled in.

Example 3a – read/write computations: $C_a(i)$ and $C_a(ii)$ are both computations of the read/write multiprocess in Example 2a, assuming that initial values of x, y and z are 0.

$$C_a(i) \begin{cases} p : \text{WRITE}(x,1) \;\; \text{WRITE}(z,2) \;\; 1 \leftarrow \text{READ}(x) \;\; 0 \leftarrow \text{READ}(y) \\ q : \text{WRITE}(y,3) \;\; 3 \leftarrow \text{READ}(y) \;\; 2 \leftarrow \text{READ}(z) \;\; 0 \leftarrow \text{READ}(x) \end{cases}$$

$$C_a(ii) \begin{cases} p : \text{WRITE}(x,1) \;\; \text{WRITE}(z,2) \;\; 1 \leftarrow \text{READ}(x) \;\; 0 \leftarrow \text{READ}(y) \\ q : \text{WRITE}(y,3) \;\; 3 \leftarrow \text{READ}(y) \;\; 0 \leftarrow \text{READ}(z) \;\; 0 \leftarrow \text{READ}(x) \end{cases}$$

Example 3b – list object computation: C_b is a computation of the list object multiprocess in Example 2b.

$$C_b \begin{cases} p : \text{APPEND}(l,4) \;\; \text{APPEND}(l,6) \;\; 4 \leftarrow \text{LAST}(l) \\ q : \text{APPEND}(l,7) \;\; \text{DELETE}(l,6) \;\; \text{DELETE}(l,4) \;\; 7 \leftarrow \text{LAST}(l) \end{cases}$$

Notice that the definition of a (system) computation does not constrain the response components of the operations in the computation. Rather, the possible responses are determined by the architecture, which we model as a set of memory consistency constraints. 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 [Lamport 1979a] 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 of a finite set, there is exactly one sequence of the elements of O that maintains this order; we denote that sequence by L.

Example 4a – Sequential Consistency on a variable multiprocess: Neither Computation $C_a(i)$ nor $C_a(ii)$ (of Example 3a) is Sequentially Consistent because it is not possible to create a sequence of all the computation's operations that both maintains program order and is valid. The following cycle in the operations of either $C_a(i)$ or $C_a(ii)$ shows that these computations are not Sequentially Consistent: $0 \leftarrow \text{READ}_p(y) \xrightarrow{validity} \text{WRITE}_q(y, 3) \xrightarrow{prog} 0 \leftarrow \text{READ}_q(x) \xrightarrow{validity} \text{WRITE}_p(x, 1) \xrightarrow{prog} 0 \leftarrow \text{READ}_p(y)$.

Example 4b – Sequential Consistency on a list object multiprocess: Computation C_b (of Example 3b) is Sequentially Consistent. One valid sequence that preserves program order is: $\text{APPEND}(l, 7)$, $\text{APPEND}(l, 4)$, $\text{APPEND}(l, 6)$, $\text{DELETE}(l, 6)$, $4 \leftarrow \text{LAST}(l)$, $\text{DELETE}(l, 4)$, $7 \leftarrow \text{LAST}(l)$.

Partial order constraints that are weaker than Sequential Consistency constitute a *weak* memory consistency model. In Subsection 4.1 we will define weak consistency models TSO and PSO and show that Computation $C_a(i)$ is PSO but not TSO and $C_a(ii)$ is TSO.

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 are 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 *trans-*

formed 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.

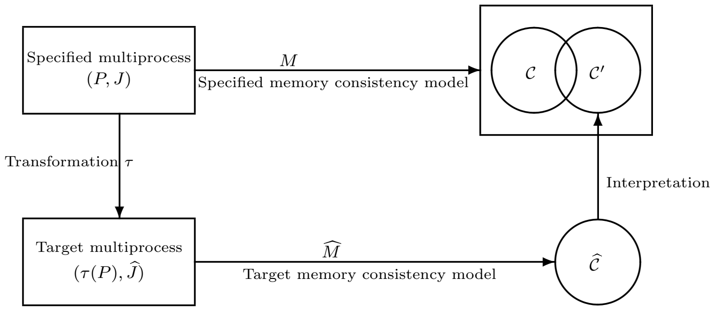

Fig. 1. Program transformation and computation interpretation.
If $\mathcal{C}' \subseteq \mathcal{C}$, then the transformation is an implementation.

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 computations produced by traveling the long way around is a non-empty subset of the computations $\mathcal{C}$ allowed by the specified system. More precisely, let $\mathcal{C}$ 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 transformation of the multiprogram P, using τ. 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 $D' \in \mathcal{C}$, 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 $D \in \mathcal{C}$ 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]

[3]Lamport [Lamport 1986b] similarly distinguishes between prescriptive and restrictive definitions of correctness of implementations.

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.*

2.3 Comparison of the Consistency Modelling Framework with other frameworks

One of the earliest frameworks, an axiomatic one by Hoare [Hoare 1972], is difficult to use to prove correctness even for simple algorithms. Owicki and Gries [Owicki and Gries 1976] provide strengthened axioms and used them to provide proofs of coordination algorithms in a parallel programming language. Misra [Misra 1986] presented a framework and applied 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 [Lamport 1979b] provided an axiomatic system and used it as the basis for descriptions and proofs of distributed system features [Lamport 1997; 1986b; 1986a]. In his extension of these axioms, Anger [Anger 1989] 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.* [Attiya et al. 1998; Attiya and Friedman 1992; Friedman 1995] 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.* [Ahamad et al. 1993; Ahamad et al. 1995; Kohli et al. 1993] 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 Merritt [Gibbons and Merritt 1992] present a framework that uses I/O Automata to represent architectural assumptions and a memory consistency model that restricts its actions.

Our framework uses ideas from much of this previous work. For example, it supports descriptions at various level of abstraction. 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 similar to that of several earlier works particularly that of Ahamad *et al.*, Lynch's I/O Automata [Lynch and Tuttle 1989; Lynch 1996], and Herlihy and Wing's Linearizability [Herlihy and Wing 1990]. There is also less-closely related work that we do not discuss here, primarily due to differences in intended objectives.

The major contribution of this paper, as we see it, is a general framework, the Consistency Modelling Framework (henceforth abbreviated CMF), for describing consistency conditions at various levels of abstraction and *proving* the equivalence between them. This is demonstrated on SPARC, a real-life example. Hence, the most closely related work to ours is Lynch and Tuttle's I/O Automata [Lynch 1996; Lynch and Tuttle 1989] and Herlihy and Wing's Linearizability [Herlihy and Wing 1990]. Both propose well-established models that support different levels of abstraction, as does CMF. I/O Automata is operational and Linearizability and CMF are non-operational. The definition style in CMF is closer to the Linearizability style. However, CMF, as presented in this paper, does not support Linearizability's global time. CMF can be extended to include a notion of global time, but this is beyond the scope of this paper.

The important difference between CMF and these two models is its generalization of the notion of computations. For Linearizability and I/O Automata, a computation is a sequence of operation invocations and responses (for the former) or events (for the latter). This is also the case with Lamport's Sequential Consistency [Lamport 1979a] (a sequence of operations). Our notion of computation, a set of sequences of operations one for each process, is more general and is more suitable for modern (loosely coupled) machines. This notion of a computation is similar to the notion of history by Ahamad *et al.*[Ahamad et al. 1993; Ahamad et al. 1995]. However, CMF is more general as it can be applied to arbitrary consistency models with arbitrary objects. Furthermore, the layering approach in CMF does not exist in the work of Ahamad *et al.*

In CMF, the notion of validity is independent from the computation. In a computation, each response (the "out" parameter) is simply glued to its operation invocation, regardless of when it is actually received by the process. This provides needed flexibility. Also, unlike some definitions (such as the complex value axiom in [Weaver and Germond 2000]), in CMF the notion of a valid sequence of operations is the natural one for the object and does not change.

Adir *et al.*'s framework [Adir et al. 2003] for Information Flow Modelling includes private registers and requires a more thorough treatment of causally-related operations (what they call the reads-from mapping). These details can be crucial (as Adir *et al.* demonstrate with the PowerPC architecture) but are not required to achieve the objective of this paper. Modelling systems at different levels of abstraction, and proving equivalence between them is not a goal of the work of Adir *et al.* and does not seem to be facilitated by their framework. It is a distinctive feature of CMF, which this paper demonstrates.

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

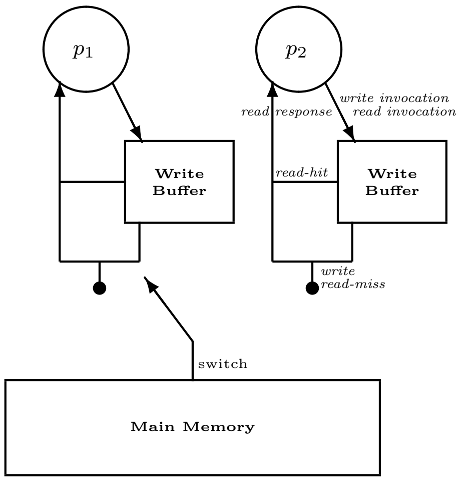

Fig. 2. Write-Buffer Architecture (for 2 processors)

is single ported with a non-deterministic switch providing one memory access at a time. Each write-buffer operates in parallel with the processor.

We contend that even the most permissive 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 weak operational constraints. A read action by some processor applied to some location can return any value written by the same processor for the same 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, which connects the buffer and main memory. Therefore, for each location and each processor, pending memory write and read operations of main memory by that processor and to that location are performed in the same order as they are issued. This means that a write that has left the buffer and is still on its way to main memory, cannot be bypassed by a read of the same location by the same processor that is issued later than the write.

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; and 3) requiring the processor to block on reads.

A partial store order write-buffer machine, which is modelled after a SPARC machine described in the version 8 manual [SPARC Int'l, Inc. 1992], 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 modelled after a SPARC version 8 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 [Kawash 2000]. 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 operations stronger than just read and write.

3.1 Partial and total store order write-buffer systems

The preceding 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 *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: To define the transformation τ 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 *out*←ACT*(objs,in)* 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. Indivisible sequences must appear to be atomic. That is, in any valid sequence of operations that is used to confirm that a computation satisfies a given memory consistency, the subsequence of operations that comprise an indivisible operation must be contiguous. For list object $\widehat{x_i}$ and variable $\widehat{x}$, define the compound operation 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 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 τ from J to $\widehat{J}$ is defined by:

```
define τ(READ_{p_i}(x))  =
        v←LAST($\widehat{x_i}$)
        if ($v = \perp$) then
            v←MEM-READ($\widehat{x}$)
        return v
define τ(WRITE_{p_i}(x, v))  =
            APPEND($\widehat{x_i}, v$)
            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 $\widehat{P} = \tau(P)$. Clearly $\widehat{P}$ is compatible with $\widehat{J}$. The program order of operations in the transformed multiprogram $\tau(P)$ is denoted $\overset{\widehat{prog}}{\longrightarrow}$.

Example 5a – write-buffer multiprogram: Transformation τ applied to the variable multiprocess in Example 2a gives the write-buffer multiprocess $(\{\widehat{p}, \widehat{q}\}, \{\widehat{x}, \widehat{y}, \widehat{z}, \widehat{x_p}, \widehat{x_q}, \widehat{y_p}, \widehat{y_q}, \widehat{z_p}, \widehat{z_q}\})$, where the multiprogram $\{\widehat{p}, \widehat{q}\}$ is as follows:

```
p̂:
  APPEND($\widehat{x_p}$, 1)
  MOVE($\{\widehat{x_p}, \widehat{x}\}$, 1)
  APPEND($\widehat{z_p}$, 2)
  MOVE($\{\widehat{z_p}, \widehat{z}\}$, 2)
  $v_1$←LAST($\widehat{x_p}$)
  if ($v_1 = \perp$) then
    $v_1$←MEM-READ($\widehat{x}$)
  return $v_1$
  $v_2$←LAST($\widehat{y_p}$)
  if ($v_2 = \perp$) then
    $v_2$←MEM-READ($\widehat{y}$)
  return $v_2$
```

```
q̂:
  APPEND($\widehat{y_q}$, 3)
  MOVE($\{\widehat{y_q}, \widehat{y}\}$, 3)
  $v_3$←LAST($\widehat{y_q}$)
  if ($v_3 = \perp$) then
    $v_3$←MEM-READ($\widehat{y}$)
  return $v_3$
  $v_4$←LAST($\widehat{z_q}$)
  if ($v_4 = \perp$) then
    $v_4$←MEM-READ($\widehat{z}$)
  return $v_4$
  $v_5$←LAST($\widehat{x_q}$)
  if ($v_5 = \perp$) then
    $v_5$←MEM-READ($\widehat{x}$)
  return $v_5$
```

Memory Consistency: Let $\widehat{O}$ be the set of all the operations of a computation $\widehat{C}$ of $(\tau(P), \widehat{J})$. Subsets of $\widehat{O}$ are denoted by $\widehat{O}|_{\mathrm{Q}}$ where Q is a collection of operation types, a process or an object. For example, $\widehat{O}|_{\text{APPEND}\cup\text{LAST}}$ is the set of all the APPEND and LAST 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 $\tau(o)$ and that $\widehat{o_i}$ and $\widehat{o_j}$ are matching operations if they correspond to the same operation. Define five partial orders on $\widehat{O}$.

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

buffer-lists: $(\widehat{o_1}, \widehat{o_2}) \in$ *buffer-lists* if and only if $\widehat{o_1} \overset{\widehat{prog}}{\longrightarrow} \widehat{o_2}$ and $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\text{APPEND}\cup\text{LAST}}$.

FIFO-per-location-memory: $(\widehat{o_1}, \widehat{o_2}) \in$ *FIFO-per-location-memory* if and only if $\widehat{o_1} \overset{\widehat{prog}}{\longrightarrow} \widehat{o_2}$ and $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\text{MEM-READ}\cup\text{MOVE}}|_{\widehat{x}}$, for some variable $\widehat{x} \in \widehat{J}$.

blocking-loads: $(\widehat{o_1}, \widehat{o_2}) \in$ *blocking-loads* if and only if $\widehat{o_1} \overset{\widehat{prog}}{\longrightarrow} \widehat{o_2}$ 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 $\widehat{o_1} \overset{\widehat{prog}}{\longrightarrow} \widehat{o_2}$ and $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\text{MOVE}}$.

Informally, matching-ops order ensures that each READ and WRITE is implemented according to the definition of its transformation. Buffer-lists order ensures that each READ or WRITE operation appears to be invoked in program order. In particular, requiring that $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\text{APPEND}\cup\text{LAST}}$ ensures that the initial part of the implementation of each READ or WRITE, which is applied to the local buffer, is applied in the program order of the corresponding READ or WRITE. Notice that READ and WRITE operations are not in general required to complete in program order since the MEM-READ and MOVE operations are not similarly constrained. FIFO-per-location-memory ensures that MOVE and MEM-READ operations by the same individual program to the same location remain in program order. 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.

Thus these orders capture the informal description of the various behavioural conditions of the write-buffer machines. The informal description of the behaviour of the partial store order write-buffer machine is captured by the first four partial orders.

$\widehat{PSO}$ consistency: A 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: A 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. Simplifying them to remove these redundancies will, in turn, simplify some of our proofs in Section 4. 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. It is straightforward to confirm that this same sequence preserves the partial orders 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 buffer-lists 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. □

Example 6a – write-buffer computations: $\widehat{C}_a(i)$ and $\widehat{C}_a(ii)$ are computations of the multiprogram in Example 5a.

$$\widehat{C}_a(i) \left\{ \begin{array}{l} \widehat{p}\text{: APPEND}(\widehat{x_p},1) \;\; \text{MOVE}(\{\widehat{x_p},\widehat{x}\},1) \;\; \text{APPEND}(\widehat{z_p},2) \;\; \text{MOVE}(\{\widehat{z_p},\widehat{z}\},2) \\ \quad 1\leftarrow\text{LAST}(\widehat{x_p}) \;\; \perp\leftarrow\text{LAST}(\widehat{y_p}) \;\; 3\leftarrow\text{MEM-READ}(\widehat{y}) \\ \widehat{q}\text{: APPEND}(\widehat{y_q},3) \;\; \text{MOVE}(\{\widehat{y_q},\widehat{y}\},3) \;\; 3\leftarrow\text{LAST}(\widehat{y_q}) \;\; \perp\leftarrow\text{LAST}(\widehat{z_q}) \\ \quad 2\leftarrow\text{MEM-READ}(\widehat{z}) \;\; \perp\leftarrow\text{LAST}(\widehat{x_q}) \;\; 0\leftarrow\text{MEM-READ}(\widehat{x}) \end{array} \right.$$

Computation $\widehat{C}_a(i)$ satisfies $\widehat{PSO}$ as is confirmed by the following sequence:

APPEND$(\widehat{x}_p,1)$, APPEND$(\widehat{z}_p,2)$, $1\leftarrow$LAST$(\widehat{x}_p)$, $\perp\leftarrow$LAST$(\widehat{y}_p)$, $0\leftarrow$MEM-READ$(\widehat{y})$, APPEND$(\widehat{y}_q,3)$, $3\leftarrow$LAST$(\widehat{y}_q)$, MOVE$(\{\widehat{z}_p,\widehat{z}\},2)$, $\perp\leftarrow$LAST$(\widehat{z}_q)$, $2\leftarrow$MEM-READ$(\widehat{z})$,

$\bot \!\leftarrow \textsc{last}(\widehat{x}_q)$, $0\leftarrow\textsc{mem-read}(\widehat{x})$, $\textsc{move}(\{\widehat{x}_p,\widehat{x}\},1)$, $\textsc{move}(\{\widehat{y}_q,\widehat{y}\},3)$.

In this sequence, $\textsc{move}(\{\widehat{z}_p,\widehat{z}\},2)$ and $\textsc{move}(\{\widehat{x}_p,\widehat{x}\},1)$ violate $\widehat{TSO}$'s FIFO-move order, which is not required for $\widehat{PSO}$.

Computation $\widehat{C}_a(i)$ does not satisfy $\widehat{TSO}$ due to the following cycle:

$$0\leftarrow\textsc{mem-read}_{\widehat{q}}(\widehat{x}) \overset{validity}{\longrightarrow} \textsc{move}_{\widehat{p}}(\{\widehat{x}_p,\widehat{x}\},1) \overset{FIFO-move}{\longrightarrow} \textsc{move}_{\widehat{p}}(\{\widehat{z}_p,\widehat{z}\},2) \overset{validity}{\longrightarrow} 2\leftarrow\textsc{mem-read}_{\widehat{q}}(\widehat{z}) \overset{blocking-loads}{\longrightarrow} 0\leftarrow\textsc{mem-read}_{\widehat{q}}(\widehat{x}).$$

$$\widehat{C}_a(ii)\begin{cases} \widehat{p}\text{: } \textsc{append}(\widehat{x_p},1)\ \ \textsc{move}(\{\widehat{x_p},\widehat{x}\},1)\ \ \textsc{append}(\widehat{z}_p,2)\ \ \textsc{move}(\{\widehat{z}_p,\widehat{z}\},2) \\ \quad 1\leftarrow\textsc{last}(\widehat{x_p})\ \ \bot\!\leftarrow\textsc{last}(\widehat{y_p})\ \ 3\leftarrow\textsc{mem-read}(\widehat{y}) \\ \widehat{q}\text{: } \textsc{append}(\widehat{y_q},3)\ \ \textsc{move}(\{\widehat{y_q},\widehat{y}\},3)\ \ 3\leftarrow\textsc{last}(\widehat{y_q})\ \ \bot\!\leftarrow\textsc{last}(\widehat{z}_q) \\ \quad 0\leftarrow\textsc{mem-read}(\widehat{z})\ \ \bot\!\leftarrow\textsc{last}(\widehat{x_q})\ \ 0\leftarrow\textsc{mem-read}(\widehat{x}) \end{cases}$$

Computation $\widehat{C}_a(ii)$ satisfies $\widehat{TSO}$ as is confirmed by the following sequence:

$\textsc{append}(\widehat{x}_p,1)$, $\textsc{append}(\widehat{z}_p,2)$, $1\leftarrow\textsc{last}(\widehat{x}_p)$, $\bot\!\leftarrow\textsc{last}(\widehat{y}_p)$, $0\leftarrow\textsc{mem-read}(\widehat{y})$, $\textsc{append}(\widehat{y}_q,3)$, $3\leftarrow\textsc{last}(\widehat{y}_q)$, $\bot\!\leftarrow\textsc{last}(\widehat{z}_q)$, $0\leftarrow\textsc{mem-read}(\widehat{z})$, $\bot\!\leftarrow\textsc{last}(\widehat{x}_q)$, $0\leftarrow\textsc{mem-read}(\widehat{x})$, $\textsc{move}(\{\widehat{x}_p,\widehat{x}\},1)$, $\textsc{move}(\{\widehat{z}_p,\widehat{z}\},2)$, $\textsc{move}(\{\widehat{y}_q,\widehat{y}\},3)$.

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 3.1) 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 (defined on page 9) from the platform (J, PSO) to the platform $(\widehat{J}, \widehat{PSO})$ and from (J, TSO) to the platform $(\widehat{J}, \widehat{TSO})$. Hence PSO and TSO, which are 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). $O|_{\textsc{write}}$ denotes the subset of O that are WRITE operations, $O|_{\textsc{read}}$ 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*[4].

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 $o_2 \in O|_{\text{WRITE}}$.

Informally, these partial orders together capture constraints on a "view of operations" as is "seen" by main memory. The same-object order ensures that the write-buffers in these models are at least FIFO per-location and the channels connecting these buffers to main memory are also FIFO. A foreign READ necessarily misses the buffer and returns a value from main memory. These READs are ordered according to when main memory "sees" them. A similar guarantee cannot be made about a domestic READ because it may hit the buffer and if it does so, it returns values that are yet to be committed to main memory. Such a READ will be indirectly "seen" by main memory in the future, only after its pending WRITE is applied to main memory. This allows domestic READs to overtake some preceding operations in program order. The following-write order captures the FIFO buffers and the fact that READs are blocking.
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.

Example 7a – TSO **and** PSO **consistency:** Computation $C_a(i)$ (of Example 3a) satisfies PSO consistency as shown by the following valid total order that preserves same-object and preceding-read orders:

$0 \leftarrow \text{READ}_p(y)$, $\text{WRITE}_p(z, 2)$, $2 \leftarrow \text{READ}_q(z)$, $0 \leftarrow \text{READ}_q(x)$, $\text{WRITE}_p(x, 1)$, $\text{WRITE}_q(y, 3)$, $1 \leftarrow \text{READ}_p(x)$, $3 \leftarrow \text{READ}_q(y)$.

However $C_a(i)$ does not satisfy TSO consistency. The following cycle shows that there is no valid total order that extends preceding-read and following-write orders:

$0 \leftarrow \text{READ}_q(x) \xrightarrow{validity} \text{WRITE}_p(x, 1) \xrightarrow{following-write} \text{WRITE}_p(z, 2) \xrightarrow{validity}$
$2 \leftarrow \text{READ}_q(z) \xrightarrow{preceding-read} 0 \leftarrow \text{READ}_q(x)$.

[4]This definition depends on the assumption that state change operations on the same object have distinct input values. If this is not the case, the definition should be changed to say a domestic READ returns a value *possibly* written by the same process. How, in general, a READ is labelled domestic or foreign is further discussed in Section 7.

Computation $C_a(ii)$ is PSO and TSO as shown by the valid total order that preserves same-object, preceding-read and following-write orders:

$0\leftarrow\text{READ}_q(z)$, $0\leftarrow\text{READ}_q(x)$, $0\leftarrow\text{READ}_p(y)$, $\text{WRITE}_q(y,3)$, $\text{WRITE}_p(x,1)$, $\text{WRITE}_p(z,2)$, $1\leftarrow\text{READ}_p(x)$, $3\leftarrow\text{READ}_q(y)$.

Note that $C_a(i)$ (respectively, $C_a(ii)$) is an interpretation of $\widehat{C}_a(i)$ (respectively, $\widehat{C}_a(ii)$).

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 ($\widehat{O}$, $\xrightarrow{\widehat{*so}}$) 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 ($\widehat{O}$, $\xrightarrow{\widehat{*so}}$). Recall that an output-generating operation $out_1\leftarrow\text{ACT}_1$*(obj,in$_1$)* and a state-change operation $out_2\leftarrow\text{ACT}_2$*(obj,in$_2$)* on the same object obj are causally related if out$_1$ = in$_2$. Because $\widehat{*so}$ is valid, any $v\leftarrow\text{LAST}_{\widehat{p_i}}(\widehat{x_i})$ 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 $v\leftarrow\text{LAST}_{\widehat{p_i}}(\widehat{x_i})$ 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 $v\leftarrow\text{LAST}_{\widehat{p_i}}(\widehat{x_i})$ 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 $v\leftarrow\text{LAST}_{\widehat{p_i}}(\widehat{x_i})$'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 $\perp\leftarrow\text{LAST}(\cdot)$ and $\text{APPEND}(\cdot,\cdot)$ and replace each $v\leftarrow\text{MEM-READ}_{\widehat{p_i}}(\widehat{x})$ with $v\leftarrow\text{READ}_{p_i}(x)$, each $v\leftarrow\text{LAST}_{\widehat{p_i}}(\widehat{x_i})$ where $v \neq\perp$ with $v\leftarrow\text{READ}_{p_i}(x)$ 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 $\widehat{o_2} = v \leftarrow \text{LAST}_{\widehat{p_i}}(\widehat{x_i})$ where $v \neq \perp$. By the validity of $\widehat{*so}$, $v \leftarrow \text{LAST}_{\widehat{p_i}}(\widehat{x_i})$ 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 $v \leftarrow \text{LAST}_{\widehat{p_i}}(\widehat{x_i})$ 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 $v \leftarrow \text{LAST}_{\widehat{p_i}}(\widehat{x_i})$. 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 sub-cases 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 sub-cases $\text{MOVE}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$ 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 sub-case 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 $o_2 \in O|_{\text{WRITE}}$, 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 replacing each $v\leftarrow$READ$_{p_i}(x)$ with ($\perp\leftarrow$LAST$_{\widehat{p_i}}(\widehat{x_i})$, $v\leftarrow$MEM-READ$_{\widehat{p_i}}(\widehat{x})$). 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. We use the notation $\widehat{T} \leftarrow \widehat{T}$ ++ o 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 $v\leftarrow$MEM-READ$(\widehat{x})$ operation that returns the value v of a MOVE$(\{\widehat{x_i}, \widehat{x}\}, v)$ such that $v\leftarrow$MEM-READ$(\widehat{x}) \xrightarrow{\widehat{T}}$ MOVE$(\{\widehat{x_i}, \widehat{x}\}, v)$[5]. For each such deleted $v\leftarrow$MEM-READ$(\widehat{x})$, replace its matching $\perp\leftarrow$LAST$(\widehat{x})$ by $v\leftarrow$LAST$(\widehat{x})$ 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 $\widehat{o_1} \xrightarrow{\widehat{S}} \widehat{o_2}$.*

[5]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 $\textsc{write}_{p_i}(x, v)$ **then**
            $\widehat{T} \leftarrow \widehat{T}$ ++ $\textsc{append}_{\widehat{p_i}}(\widehat{x_i}, v)$ ++ $\textsc{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v))$
        **if** $o$ is $v{\leftarrow}\textsc{read}_{p_i}(x)$ **then**
            $\widehat{T} \leftarrow \widehat{T}$ ++ $\bot{\leftarrow}\textsc{last}_{\widehat{p_i}}(\widehat{x_i})$ ++ $v{\leftarrow}\textsc{mem-read}_{\widehat{p_i}}(\widehat{x}))$
        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 $\textsc{write}_{p_i}(x, v)$ **then**
            $\widehat{T} \leftarrow \widehat{T}$ ++ $\textsc{append}_{\widehat{p_i}}(\widehat{x_i}, v)$
        **if** $o'$ is $v{\leftarrow}\textsc{read}_{p_i}(x)$ **then**
            $\widehat{T} \leftarrow \widehat{T}$ ++ $\bot{\leftarrow}\textsc{last}_{\widehat{p_i}}(\widehat{x_i})$ ++ $v{\leftarrow}\textsc{mem-read}_{\widehat{p_i}}(\widehat{x})$
        Record $o'$ as marked
        Advance $\downarrow_{p_i}$
    **else** ( $\downarrow_{*so} = o$ and $o$ is marked)
        **if** $o$ is a $\textsc{write}_{p_i}(x, v)$ **then**
            $\widehat{T} \leftarrow \widehat{T}$ ++ $\textsc{move}_{\widehat{p_i}}(\{\widehat{x_i}, \widehat{x}\}, v)$
        Advance $\downarrow_{*so}$
```

Fig. 3. Construction of $\widehat{T}$

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, $v{\leftarrow}\textsc{read}_{p_j}(x)$, in $*so$. Call this READ r. It returns the value of the most recent preceding WRITE operation to the same object, $\textsc{write}_{p_i}(x, v)$. Call this WRITE w. We now examine the validity of the LAST and MEM-READ operations that are placed onto $\widehat{S}$ and correspond to r.

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 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 $\perp\!\!\leftarrow$LAST$(\widehat{x})$ and $v\leftarrow$MEM-READ$(\widehat{x})$ 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 pair of LAST and MEM-READ operations.

The $v\leftarrow$MEM-READ$(\widehat{x})$ 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 $\perp\!\!\leftarrow$LAST$(\widehat{x})$ 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 occurred 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 3.1 are equivalent to these specifications in the SPARC architecture manuals [SPARC Int'l, Inc. 1992; Weaver and Germond 2000]. The SPARC version 8 manual defines total store order and partial store order [SPARC Int'l, Inc. 1992]. The version 9 manual [Weaver and Germond 2000] 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.

Subsection 5.1 defines partial store order and total store order architecture systems within our framework using a validity condition and a program order definition that mimic those used in the version 9 architecture manual [Weaver and Germond 2000]. In Subsection 5.2, we argue that our architectural level definition captures that of the SPARC manuals. Finally, in Subsection 5.3, we prove that the partial and total store order write-buffer systems of Subsection 3.1 exactly implement the partial and total store order architecture systems (respectively). Throughout, we restrict our scope 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 [Kawash 2000], but are outside the scope of this paper.

5.1 Systems based on SPARC manual specifications

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 Consistency Modelling Framework's notion of programs, objects, and memory consistency. In the SPARC architecture manual, programs are described in terms of the objects they operate on, and the validity of these objects is defined in terms of the programs. So, in order to remain close to the manual specifications, we begin our definition with programs, even though there is a forward reference to objects.

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 $\overset{prog}{\longrightarrow}$.

Objects: A SPARC *variable* x is the set of sequences over operations of the type $\text{WRITE}_{p_i}(x, \cdot)$ and $\Delta \leftarrow \text{READ}_{p_j}(x)$, $p_i, p_j \in P_a$, 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 $\text{WRITE}_{p_j} \overset{prog}{\longrightarrow} \text{READ}_{p_i}$.
> 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 $o_1 \overset{prog}{\longrightarrow} o_2$, $\exists x \in J_a$, $o_1 \in O_a|x$, and $o_2 \in O_a|_{\text{WRITE}}|x$.

SPARC-*read:* $(o_1, o_2) \in$ SPARC-read if and only if $o_1 \overset{prog}{\longrightarrow} o_2$ and $o_1 \in O_a|_{\text{READ}}$.

SPARC-*write:* $(o_1, o_2) \in$ SPARC-write if and only if $o_1 \overset{prog}{\longrightarrow} o_2$ and $o_2 \in O_a|_{\text{WRITE}}$.

Define two 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

The purpose of this subsection is to argue that the systems (P_a, J_a, PSO_a) and (P_a, J_a, TSO_a) just defined using the Consistency Modelling Framework are equivalent to the systems defined in the SPARC Architecture Manual version 9 [Weaver and Germond 2000]. This is done by showing where each part of the definition is captured in the manual. Of necessity, this subsection relies heavily on this manual, especially Appendix D: Formal Specifications of the Memory Model. Thus, this subsection is not self-contained. It can be skipped however, without jeopardizing understanding of the rest of the paper. The 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 $P_a = \{p_1, \ldots p_n\}$. 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 [Weaver and Germond 2000]). Thus, SPARC program order agrees with the program order defined in our framework.

Objects: The validity condition for SPARC variables is more complex than the validity condition of variables defined in Section 2 of this paper; it is based directly on the manual definition of Section D.4.5. "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 [Weaver and Germond 2000].) It is straightforward to check that the definition of SPARC variables is exactly this condition.

Memory Consistency: Sections D.5 and D.6 specify the partial and total store orders that we call PSO_a and TSO_a. These specifications, however, rely on the RMO (relaxed memory order) specification of Section D.4. They also rely on definitions of MEMBAR instructions. A MEMBAR instruction is not an operation; rather it is a memory barrier that enforces constraints on what instructions can be reordered. There are four basic MEMBAR instructions. A MEMBAR `#LoadStore` imposes the constraint that any instruction with load semantics that is before the MEMBAR in program order, must be completed before any instruction with store semantics that is after the MEMBAR can be invoked. MEMBAR `#LoadLoad`, MEMBAR `#StoreLoad`, and MEMBAR `#StoreStore` are defined similarly. Basic MEMBARs can be combined; for example the constraints imposed by MEMBAR `#LoadLoad|#LoadStore` is the union of the constraints of MEMBAR `#LoadLoad` and MEMBAR `#LoadStore`. Our 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 is invoked. 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.

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.

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. Transformation τ

refers to the transformation with this name in Subsection 3.1. The key difference between this proof and those in Section 4 is that a new construction is used to show that τ is a compiler. In the new construction, no re-ordering is required; the following conversions are applied: all LAST operations that return non-$\perp$ and MEM-READs are replaced by READ operations, and MOVEs are replaced by WRITEs. Construction two is used to establish that 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.

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 [Weaver and Germond 2000], 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 [Compaq Computer Corp. 1998] as formalized by Attiya and Friedman [Attiya and Friedman 1994]. We will argue that none of these memory consistency models have an exact implementation on any natural write-buffer multiprocessor. The key is a Coherent computation (defined below) that cannot occur on even the most permissive write-buffer machine we might imagine. So, we first define this very basic write-buffer system.

6.1 Basic write-buffer system

A set object is similar to a list object but lacks order.

A *set object* s, is the set of sequences over operations of the type INSERT$(s,\cdot)$, DELETE$(s,\cdot)$, and $\Delta \leftarrow$SELECT(s) that satisfy the validity condition:

> For any SELECT operation on set object, s, it returns a value $\rho \neq \perp$ if and only if there is a preceding INSERT(s,ρ) operation and, between this INSERT and the SELECT there is no DELETE(s,ρ). Otherwise, it returns $\perp$.[6]

[6]This definition exploits the assumption that input values are distinct.

The basic write-buffer platform is defined similarly to partial store order and total store order write-buffer platform except that $\widehat{J}$ contains set objects (instead of list objects). The result is a machine that admits many more computations than the partial store order write-buffer machine.

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: For set object or list object $\widehat{x_i}$ and variable $\widehat{x}$, define the compound operation $\textsc{move}(\{\widehat{x_i}, \widehat{x}\}, v)$ to be the indivisible sequence of operations: $\textsc{delete}(\widehat{x_i}, v)$,$\textsc{write}(\widehat{x}, v)$. A sequence of operations on set 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 γ(READ_{p_i}(x)) =
                        v←SELECT(x̂_i)
                        if (v =⊥) then
                           v←MEM-READ(x̂)
                        return v
define γ(WRITE_{p_i}(x, v)) =
                           INSERT(x̂_i, v)
                           MOVE({x̂_i, x̂}, 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 multiprogram $\gamma(P)$ is denoted $\widehat{\xrightarrow{prog}}$.

Memory Consistency: Let $\widehat{O}$ be the set of all the operations of a computation $\widehat{C}$ of $(\gamma(P), \widehat{J})$. 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. The following three partial orders on the set operations in $\widehat{O}$ are analogous to the first three partial orders defined for list operations in Subsection 3.1.

matching-ops: $(\widehat{o_1}, \widehat{o_2}) \in$ *matching-ops* if and only if $\widehat{o_1} \widehat{\xrightarrow{prog}} \widehat{o_2}$ 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 $\widehat{o_1} \widehat{\xrightarrow{prog}} \widehat{o_2}$ and $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\textsc{insert} \cup \textsc{select}}$.

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

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.

Notice that in a basic write-buffer machine, a READ can return the value of any of the previous WRITEs to the same location by the same processor that has not been moved to main memory, not necessarily the latest one. This may seem unrealistic. This excessively permissive definition, however, serves to strengthen our impossibility claims in the Subsection 6.3.

6.2 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 [Frigo 1998]. Coherence only requires the same-object order of Subsection 4.1. 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 same-object order.

According to the Alpha architecture specifications [Compaq Computer Corp. 1998], 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 [Compaq Computer Corp. 1998]). From the Alpha definition by Attiya and Friedman [Attiya and Friedman 1994], 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 [Weaver and Germond 2000] allows some computations that are not Coherent. The third "memory order constraint" (page 260 of [Weaver and Germond 2000]), 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 [Weaver and Germond 2000]) 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 for some $x \in J$ $o_1, o_2 \in O|_x$ and $(o_1 \in O|_{\text{WRITE}}$ or $o_2 \in O|_{\text{WRITE}})$.

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.

6.3 Coherence, Alpha and RMO using write-buffers

Consider the following multiprogram, 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 this multiprogram:

$$C_{\overline{WB}} \begin{cases} p : 5{\leftarrow}\text{READ}(x) \;\; \text{WRITE}(x,1) \;\; 2{\leftarrow}\text{READ}(x) \;\; \text{WRITE}(y,6) \;\; \text{WRITE}(y,4) \\ q : 6{\leftarrow}\text{READ}(y) \;\; \text{WRITE}(y,3) \;\; 4{\leftarrow}\text{READ}(y) \;\; \text{WRITE}(x,5) \;\; \text{WRITE}(x,2) \end{cases}$$

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

$$S_x = \text{WRITE}_q(x,5),\; 5{\leftarrow}\text{READ}_p(x),\; \text{WRITE}_p(x,1),\; \text{WRITE}_q(x,2),\; 2{\leftarrow}\text{READ}_p(x).$$

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

$$S_y = \text{WRITE}_p(y,6),\; 6{\leftarrow}\text{READ}_q(y),\; \text{WRITE}_q(y,3),\; \text{WRITE}_p(y,4),\; 4{\leftarrow}\text{READ}_q(y).$$

These valid total orders confirm that $C_{\overline{WB}}$ is Coherent. Notice that all READ operations in $C_{\overline{WB}}$ are foreign READ operations. Also observe that $C_{\overline{WB}}$ 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* $C_{\overline{WB}}$.

PROOF. Suppose there is a computation $\widehat{C}$ of $(\widehat{P}, \widehat{K}, \widehat{WB})$ whose interpretation is $C_{\overline{WB}}$. Because each READ in $C_{\overline{WB}}$ 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: $\perp{\leftarrow}\text{SELECT}_{\widehat{u}}(\widehat{z_u})$, $v{\leftarrow}\text{MEM-READ}_{\widehat{u}}(\widehat{z})$.

Since $\widehat{C}$ is a basic write-buffer computation, 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 $\rightsquigarrow$.

To ensure the validity of the two SELECT operations in dashed boxes, each must be preceded by the MOVE operation by the same process in the solid box since otherwise, SELECT cannot return $\perp$. In addition, to ensure validity of the $5{\leftarrow}\text{MEM-READ}_{\widehat{p}}(\widehat{x})$ and $6{\leftarrow}\text{MEM-READ}_{\widehat{q}}(\widehat{y})$ 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:

$$\begin{tikzcd}
\perp\!\leftarrow\!\textsc{select}_{\widehat{p}}(\widehat{x_p}) \arrow[r, dashed] \arrow[d] & \textsc{insert}_{\widehat{p}}(\widehat{x_p},1) \arrow[r, dashed] \arrow[d] & \boxed{\perp\!\leftarrow\!\textsc{select}_{\widehat{p}}(\widehat{x}_p)} \arrow[r, dashed] \arrow[d] & \textsc{insert}_{\widehat{p}}(\widehat{y}_p,6) \arrow[r, dashed] \arrow[d] & \textsc{insert}_{\widehat{p}}(\widehat{y}_p,4) \arrow[d] \\
5\!\leftarrow\!\textsc{mem-read}_{\widehat{p}}(\widehat{x}) \arrow[r, squiggly] & \boxed{\textsc{move}_{\widehat{p}}(\{\widehat{x_p},\widehat{x}\},1)} \arrow[r, squiggly] & 2\!\leftarrow\!\textsc{mem-read}_{\widehat{p}}(\widehat{x}) & \textsc{move}_{\widehat{p}}(\{\widehat{y}_p,\widehat{y}\},6) \arrow[r, squiggly] & \textsc{move}_{\widehat{p}}(\widehat{y},4)
\end{tikzcd}$$

$$\begin{tikzcd}
\perp\!\leftarrow\!\textsc{select}_{\widehat{q}}(\widehat{y}_q) \arrow[r, dashed] \arrow[d] & \textsc{insert}_{\widehat{q}}(\widehat{y}_q,6) \arrow[r, dashed] \arrow[d] & \boxed{\perp\!\leftarrow\!\textsc{select}_{\widehat{q}}(\widehat{y}_q)} \arrow[r, dashed] \arrow[d] & \textsc{insert}_{\widehat{q}}(\widehat{x_q},5) \arrow[r, dashed] \arrow[d] & \textsc{insert}_{\widehat{q}}(\widehat{x_q},2) \arrow[d] \\
6\!\leftarrow\!\textsc{mem-read}_{\widehat{q}}(\widehat{y}) \arrow[r, squiggly] & \boxed{\textsc{move}_{\widehat{q}}(\{\widehat{y}_q,\widehat{y}\},3)} \arrow[r, squiggly] & 4\!\leftarrow\!\textsc{mem-read}_{\widehat{q}}(\widehat{y}) & \textsc{move}_{\widehat{q}}(\{\widehat{x_q},\widehat{x}\},5) \arrow[r, squiggly] & \textsc{move}_{\widehat{q}}(\widehat{x},2)
\end{tikzcd}$$

Fig. 4. Orders that must be extended in $\widehat{WB}$

$\textsc{move}_{\widehat{p}}(\{\widehat{y}_p,\widehat{y}\},6)$, $6\!\leftarrow\!\textsc{mem-read}_{\widehat{q}}(\widehat{y})$, $\textsc{move}_{\widehat{q}}(\{\widehat{y}_q,\widehat{y}\},3)$, $\perp\!\leftarrow\!\textsc{select}_{\widehat{q}}(\widehat{y}_q)$, $\textsc{insert}_{\widehat{q}}(\widehat{x_q},5)$, $\textsc{move}_{\widehat{q}}(\{\widehat{x_q},\widehat{x}\},5)$, $5\!\leftarrow\!\textsc{mem-read}_{\widehat{p}}(\widehat{x})$, $\textsc{move}_{\widehat{p}}(\{\widehat{x_p},\widehat{x}\},1)$, $\perp\!\leftarrow\!\textsc{select}_{\widehat{p}}(\widehat{x_p})$, $\textsc{insert}_{\widehat{p}}(\widehat{y}_p,6)$, $\textsc{move}_{\widehat{p}}(\{\widehat{y}_p,\widehat{y}\},6)$.
Since the union of the partial orders cannot be extended into a valid total order that preserves the basic write-buffer orders, computation $\widehat{C}$ does not exist. □

Computation $C_{\overline{WB}}$ establishes that some particular platforms cannot be exactly implemented on any write-buffer machine.

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. $C_{\overline{WB}}$ 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, $C_{\overline{WB}}$ is also an RMO computation. That is, $C_{\overline{WB}}$ is a computation that could occur on any Coherent, RMO or Alpha platform. However, by Claim 6.1, $C_{\overline{WB}}$ is not the interpretation of any computation of any multiprogram when transformed to basic write-buffer system. Hence, $C_{\overline{WB}}$ cannot be the interpretation of any write-buffer system computation. □

Since there is a computation that is Coherent, RMO and Alpha but does not satisfy even basic write-buffer consistency, the various write-buffer consistencies are either stronger than or incomparable to Coherence, RMO and Alpha. To examine this further, first consider Coherence. Any PSO computation has a valid total order that extends same object order, so any PSO computation is Coherent. Since, the partial store order write-buffer machine is an exact compiler for a PSO platform, we conclude that $\widehat{PSO}$ (and hence $\widehat{TSO}$) consistency is strictly stronger than Coherence. RMO is weaker than Coherence, so $\widehat{PSO}$ is also strictly stronger than RMO. For Alpha the case is more involved. Alpha consistency is stronger than Coherence because any Alpha computation must have a valid total order that preserves control and data dependencies as well as Coherence. The framework as presented in this paper does not capture dependencies that arise from individual programs and the private registers of processes. However, we could imagine a write-buffer platform that also imposes the register, control and data dependencies that are required by Alpha. Such a machine, with $\widehat{PSO}$ consistency for memory operations would be strictly stronger than Alpha consistency. Equivalently, $\widehat{PSO}$ is strictly stronger than the consistency of Alpha when the additional constraints imposed by each process' local actions on its private registers and the control structure of its program are ignored. The details required to capture the full Alpha consistency are beyond the scope of this paper.

Weakening the partial store order write-buffer machine to a basic write-buffer machine, changes the relationship of the machine with Coherence, Alpha, and RMO from strictly stronger to incomparable. Consider the trivial multiprocess $(\{s\}, \{x\})$ with just one process and one variable, where the program for s is:

$s : \text{WRITE}(x, 1)\ \ \text{WRITE}(x, 2)\ \ \text{READ}(x)$

and the computation of the multiprocess $(\{\gamma(s)\}, \{\gamma(x)\})$ is:

$C1 \qquad \gamma(s) : \text{INSERT}(\widehat{x_s}, 1)\ \ \text{MOVE}(\{\widehat{x_s}, \widehat{x}\}, 1)\ \ \text{INSERT}(\widehat{x_s}, 2)$
$\text{MOVE}(\{\widehat{x_s}, \widehat{x}\}, 2)\ \ 1 \leftarrow \text{SELECT}(\widehat{x_s}).$

The following is a valid sequence of all the operations of $C1$ that preserves matching-ops order, buffer-sets order, and FIFO-per-location-memory order, assuming x is initialized to 0:
$\text{INSERT}(\widehat{x_s}, 1),\ \text{INSERT}(\widehat{x_s}, 2),\ 1 \leftarrow \text{SELECT}(\widehat{x_s}),\ \text{MOVE}(\{\widehat{x_s}, \widehat{x}\}, 1),\ \text{MOVE}(\{\widehat{x_s}, \widehat{x}\}, 2).$
Thus, $C1$ is a computation of the basic write-buffer machine. The interpretation of this computation is $C2$:

$C2 \qquad s : \text{WRITE}(x, 1)\ \ \text{WRITE}(x, 2)\ \ 1 \leftarrow \text{READ}(x)$

which is clearly not Coherent. Since Computation $C_{\overline{WB}}$ is Coherent but not $\widehat{WB}$ and $C1$ is $\widehat{WB}$ but not Coherent, Coherence and $\widehat{WB}$ are incomparable. Furthermore, for $C2$, program order, same-object order and weak-same-object order are all the same. Thus $C2$ is also not RMO.

Notice that Computation $C1$ does not satisfy the buffer-lists order. Consider a consistency model that is stronger than $\widehat{WB}$ but weaker than $\widehat{PSO}$:

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

In fact, using a proof similar to that in Lemma 4.4, it can be shown that any $\widehat{list - WB}$ computation is Coherent. To summarize:

Theorem 6.3.
—*The write-buffer machine that delivers $\widehat{list - WB}$ consistency is stronger than Alpha (without register and control dependencies) or Coherence or RMO consistency.*
—*The write-buffer machine that delivers $\widehat{WB}$ consistency is incomparable to Alpha or Coherence or RMO consistency.*

7. SUMMARY AND FURTHER COMMENTS

This 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.

The paper provides simple memory consistency specifications for the SPARC version 8 architecture variants, total and partial store orders. The framework is used to show 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 minimum consistency guarantee that would arise from any reasonable implementation on a machine with write-buffers of a multiprogram that uses shared variables is defined. This guarantee is used to prove that each of several well known memory consistency models does not exactly correspond to any implementation that uses write-buffers. For instance, RMO as described in the official SPARC version 9 manual [Weaver and Germond 2000] cannot be a description of a write-buffer machine. We conjecture that the manual description was not faithful to the intended operation of an RMO machine. As another example, the memory consistency of the Alpha multiprocessor [Compaq Computer Corp. 1998] as formalized by Attiya and Friedman [Attiya and Friedman 1994] 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 [Compaq Computer Corp. 1998]). However, we have shown that any such utilization of write-buffers would necessarily give a memory consistency model more constrained than that of Alpha.

The assumption of distinct input values for state-change operations is for convenience and is often assumed in memory consistency modelling. In this paper it simplifies the definitions of set and list objects considerably. The more general definitions would add messiness to the already detailed proofs without adding insights. Notice that determining whether a read is “foreign” or “domestic” is easy if written values do happen to be distinct, but this is not part of the definition of foreign/domestic reads. Whether or not some reads are foreign could possibly be inferred in other ways by analyzing the program even when written values are not distinct. For example, reads of a shared single-writer variable by all other processes are necessarily foreign. When it cannot be asserted that a read is foreign, programmers must allow for the case that the read may be domestic and therefore ensure their code is correct under the additional re-orderings that might arise. For chip testing and verification purposes, which is one area where we envision this work to be applicable, unique values can be enforced by augmenting each value with a unique Lamport time-stamp [Lamport 1978].

Many concerns such as asynchrony, weak consistency and faults make implementation of correct distributed systems very difficult. Writing correct programs is facilitated by providing the parallel and distributed application developers with definitions at the level of the programming instructions. Elsewhere [Higham and Kawash 2000; Kawash 2000; Higham and Kawash 2005] we exploited the non-operational definitions for TSO and PSO developed in this paper to show some algorithmic possibilities and impossibilities for these architectures. For example,

—Contrary to Sequential Consistency, there is no solution for mutual exclusion on a TSO or PSO machine using only variables, so expensive synchronization primitives are essential.

—There is a (non-waitfree) construction of a producer-consumer queue in a TSO or PSO system, using only reads and writes of variables for any number of producers and consumers.

—One read/write (multi-writer) variable is necessary and sufficient for building a one producer and one consumer queue in a TSO or PSO system.

—There are wait-free solutions for a two-process producer-consumer queue that use only single-writer variables in a TSO or PSO system.

The task of reasoning about the correctness of distributed systems is further eased if programmers can assume a strong consistency model such as Sequential Consistency or Linearizability. This motivates us to hide the consistency model complexities by building compilers that transform a program that is correct under Sequential Consistency to an equivalent program that is also correct under weaker consistency models. 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.

Acknowledgments

We gratefully thank three anonymous referees for their valuable insights and suggestions that substantially improved the content and presentation of the paper. We also thank the Natural Sciences and Engineering Research Council of Canada (NSERC); this research was supported in large part by NSERC grant number OG-POO41900.

REFERENCES

Adir, A., Attiya, H., and Shurek, G. 2003. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst. 14,* 5, 502–515.

Ahamad, M., Bazzi, R., John, R., Kohli, P., and Neiger, G. 1993. The power of processor consistency. In *Proc. 5th Int'l Symp. on Parallel Algorithms and Architectures.* 251–260. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.

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

Anger, F. 1989. On Lamport's interprocessor communication model. *ACM Trans. on Programming Languages and Systems 11*, 404–417.

Attiya, H., Chaudhuri, S., Friedman, R., and Welch, J. 1998. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. *SIAM Journal of Computing 27,* 1 (February), 65–89.

Attiya, H. and Friedman, R. 1992. A correctness condition for high performance multiprocessors. In *Proc. 24th Int'l Symp. on Theory of Computing.* 679–690.

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

Compaq Computer Corp. 1998. *The Alpha Architecture Handbook*. Compaq Computer Corporation. Order number: EC-QD2KC-TE.

Dubois, M., Scheurich, C., and Briggs, F. 1986. Memory access buffering in multiprocessors. In *Proc. 13th Int'l Symp. on Computer Architecture*. 434–442.

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

Frigo, M. 1998. The weakest reasonable memory model. M.S. thesis, Department of Electrical Engineering and Computer Science, MIT.

Gibbons, P. and Merritt, M. 1992. Specifying nonblocking shared memories. In *Proc. 4th Int'l Symp. on Parallel Algorithms and Architectures*. 306–315.

Gontmakher, A. and Schuster, A. 2000. Java consistency: nonoperational characterizations for Java memory behavior. *ACM Trans. on Computer Systems 18,* 4, 333–386.

Goodman, J. 1989. Cache consistency and sequential consistency. Tech. Rep. 61, IEEE Scalable Coherent Interface Working Group. March.

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

Higham, L. and Kawash, J. 1998. Java: Memory consistency and process coordination (extended abstract). In *Proc. 12th Int'l Symp. on Distributed Computing, Lecture Notes in Computer Science volume 1499*. 201–215.

Higham, L. and Kawash, J. 2000. 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*. 355–366.

Higham, L. and Kawash, J. 2005. Process coordination in the absence of sequential consistency. In Preparation.

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

Intel Corp. 2002. Intel itanium architecture software developer's manual, volume 2: System architecture. http://www.intel.com/.

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

Kawash, J. 2000. Limitations and capabilities of weak memory consistency systems. Ph.D. thesis, Department of Computer Science, The University of Calgary.

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

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

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

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

Lamport, L. 1986a. The mutual exclusion problem (parts I and II). *Journal of the ACM 33,* 2 (April), 313–326 and 327–348.

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

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

Lynch, N. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Lynch, N. and Tuttle, M. 1989. An introduction to input/output automata. *CWI Quarterly 2,* 3 (September), 219–246.

Misra, J. 1986. Axioms for memory access in asynchronous hardware systems. *ACM Trans. on Programming Languages and Systems 8,* 1, 142–153.

Owicki, S. and Gries, D. 1976. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM 19,* 5 (May), 279–285.

Park, S. and Dill, D. 1999. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers 48,* 2 (February), 227–235.

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

Sun Microsystems. 2004. http://www.sun.com/processors/whitepapers/us4_whitepaper.pdf.

Weaver, D. and Germond, T., Eds. 1994-2000. *The SPARC Architecture Manual version 9.* Prentice-Hall. http://developers.sun.com/solaris/articles/sparcv9.pdf.

A. PROOF OF THEOREM 5.1

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 $(\tau(P_a), \widehat{J}, \widehat{TSO})$[7]. 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 $\perp\!\leftarrow$LAST$(\cdot)$ and each APPEND$(\cdot,\cdot)$ and replace each $v\leftarrow$MEM-READ$_{\widehat{p_i}}(\widehat{x})$ with $v\leftarrow$READ$_{p_i}(x)$, each $v\leftarrow$LAST$_{\widehat{p_i}}(\widehat{x_i})$ where $v \neq \perp$ with $v\leftarrow$READ$_{p_i}(x)$ 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 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 $\widehat{m'}$

[7]Where τ is the transformation defined in subsection 3.1, used here to transform from J_a to $\widehat{J}$.

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 $o_2 \in O_a|_{\text{WRITE}}|x$ 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 $o_2 \in O_a|_{\text{WRITE}}$, 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, $v \leftarrow \text{READ}_{p_j}(x)$, 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 $\perp\!\leftarrow$LAST$(\widehat{x})$ and $v\leftarrow$MEM-READ$(\widehat{x})$ 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 $v\leftarrow$MEM-READ$(\widehat{x})$ 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 $\perp\!\leftarrow$LAST$(\widehat{x})$ 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 C 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. $\square$

...