T D C C

P F

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
faturu@csd.uoc.gr

Programmer-Centric Memory Consistency Modelling

Lisa Higham* Jalal Kawash† Abhijeet Pareek‡

Abstract

A framework for modelling memory consistency is presented. The framework is used to specify the operation of a total-store-order write buffer multiprocessor machine. The framework is used again to specify a more abstract "programmer-centric" model that does not refer to the write-buffer architecture. Finally, we prove that the abstract model correctly captures the computations of the operational model. A mutual exclusion algorithm is used

*University of Calgary, higam@ucalgary.ca
†University of Calgary, jkawash@ucalgary.ca
‡University of Calgary, apareek@ucalgary.ca

Bulletin of the EATCS no 105, pp. 118–140, October 2011

to illustrate the advantage of the programmer-centric memory consistency models and the impact on program correctness of instruction reordering resulting from hardware and software optimizations.

1 Introduction

Researchers in theoretical computer science who design algorithms for asynchronous multiprocessor systems typically assume that the system is sequentially consistent. This means, in Lamport's words, that "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."[12]

Modern multiprocess systems however do not provide this guarantee. Multicore machines, grid configurations, and in general any distributed processing system that uses multiple processors typically uses complex memory and communication structures to enhance performance. These may include write-buffering with read by-passing, multi-level caching, distributed shared address spaces, and multiple and hierarchical buses. A consequence of these enhancements is complicated memory behavior, which results in outcomes that are not sequentially consistent. Programs designed for sequentially consistent systems will need additional synchronization to remain correct on these machines, which produce executions with weaker guarantees than sequential consistency.

A multiprocessor computer that associates a write-buffer with each processor provides a simple example. Figure 1 illustrates a write-buffer architecture for 2 processors. The main memory is single ported with a non-deterministic switch providing one memory access at a time. When a processor performs a write it stores the write in the write-buffer and then proceeds with its program. The write-buffer 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 the *most recent* 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. Because each channel between a processor and an individual main memory location is FIFO, 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.

There are several variants of this basic architecture depending on additional assumptions. In this paper, we make two assumptions, which correspond to the SPARC total-store-order machine [19, 20]. (1) Reads are blocking, meaning that a

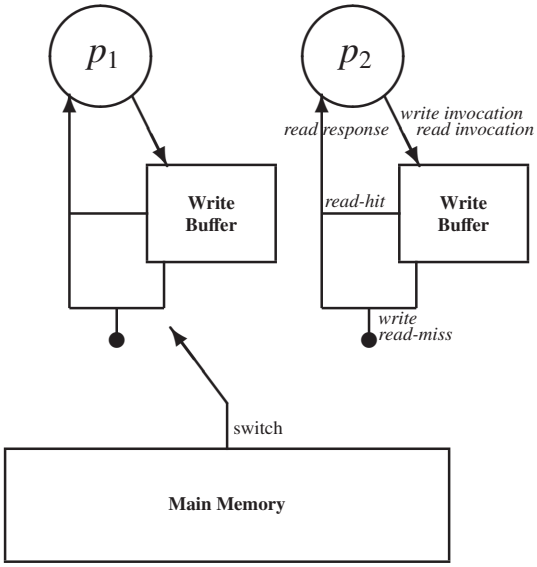


Figure 1: Write-Buffer Architecture (for 2 processors)

read blocks the processor from issuing further instructions until the read completes by returning a value. (2) The buffer is emptied in FIFO order.

Adding write-buffers improves the overall multiprocessor performance by hiding write latencies and improving processor-level parallelism. However, it complicates reasoning about concurrent (parallel or multi-threaded) programs, and ensuring the correctness of such program becomes more challenging.

For a simple example, consider Peterson's two-process mutual exclusion algorithm [17], reproduced in Algorithm 1. This algorithm is known to be correct for sequentially consistent systems.

Now imagine that the two processes in Algorithm 4 run concurrently at approximately the same speed on separate processors of a total-store-order machine. For each process $\iota \in \{0, 1\}$, ι issues the write to flag[ι] (line 1), which is sent to ι's write-buffer, and then issues its write of $\bar{\iota}$ to turn (line 2), which is also sent to ι's write-buffer. Thus, each process ι could perform the test loop at line 3 before $\bar{\iota}$'s write-buffer commits the write of flag[$\bar{\iota}$] to main memory. So each $\iota \in \{0, 1\}$ will read false for flag[$\bar{\iota}$] and hence both processes enter the critical section simultaneously breaking the safety property.

In fact, it is impossible to solve mutual exclusion on a total-store-order machine with only reads and writes of variables [6, 11]. That is, more powerful synchronization primitives must be used in order to ensure the correctness of Peterson's algorithm on SPARC. In Section 4, we show how to do this using *barrier* instructions, the function of which is to enforce additional orderings on the execution of reads and writes.

This example illustrates a simple case of a pervasive phenomenon: textbook correctness, which assumes sequential consistency is not enough. Additional syn-

Algorithm 1: Peterson's two-process mutual exclusion algorithm

```
Shared variables:
      flag [0...1] ∈ {true, false}, initialized to false
      turn ∈ {0, 1}

Program for process ι (ι ∈ {0, 1}, and ῑ = 1 − ι)

entry:
1     flag [ι] ← true
2     turn ← ῑ
3     while flag [ῑ] and turn = ῑ do nothing

⟨ Critical Section ⟩

exit:
4     flag [ι] ← false
```

chronization is necessary to ensure correctness on real systems. But, since additional synchronization incurs a huge performance hit, programmers strive to add only the necessary synchronization, rather than sprinkling it "everywhere". To do this, programmers need to understand exactly what computations can arise from their programs when run on a particular multiprocessor or multicore machine. Asking programmers to reason in terms of the lower level operations of each particular machine, however, seems unreasonable. For example, in the case of the total-store-order machine, reasoning about when writes enter and leave the write-buffer, and whether reads return from the buffer or from main memory, is complicated and error prone. Instead, we advocate for a what we call a *programmer-centric* specification of the computations that can arise from a given program on a given machine. By this we mean a description in terms of the instructions used by the programmer. It does not refer to the low level events imposed by the hardware when it executes these instructions. Programmers have a simple abstraction that captures the outcome of any computation on a sequentially consistent system. We aim for a similarly abstract and intuitive model for any machine. That is, our approach specifies a weak memory model as a natural generalization of sequential consistency. This approach uses enough formalism to provide precision, conciseness, and to ensure no ambiguities. Nevertheless, it is accessible to professional system programmers since the mathematical underpinnings are straightforward. Finally, the framework used for our programmer-centric specifications is flexible: it can be used to specify machines and systems at various levels of abstraction.

Proving the equivalence of an operational description of a machine and its more abstract programmer-centric model is facilitated by expressing both in the same framework.

Our framework is presented in in Section 2.

In Section 3 we apply our framework and our proof techniques to the total-store order machine. First we convert the informal description above into an operational specification in terms of our framework. Then we give a corresponding non-operational programmer-centric specification also using the framework. Of course, any such specification needs to be proved correct, meaning that any outcome of a program that could happen when the program is run on the total-store-order machine is included in the set of outcomes of the programmer-centric specification for the same program. In this paper we give a complete proof of this. The result is a generalization of an earlier proof [5] but it is extended and restructured using logic diagrams to achieve the proof obligations.

Imagine a machine where all shared memory is accessed by a single switch through which each process must queue. There is only one bus, and there is no replicated memory such as in local caches or write-buffers. In such a machine, if each process issues one instruction at a time in the order specified by its program, the outcome would be sequentially consistent. It is well known that even in this setting, however, instruction reorderings can arise from compiler optimizations or CPU speculative execution, read pre-fetching, and pipelining. These reorderings are independent of the interconnection hardware between the processors and the shared memory. Such reorderings are safe in a uniprocessor environment because they cannot alter the correctness of a sequential program; they are not safe in a concurrent environment. Figure 2 reproduces a classic simple example [4].

	Process 1	Process 2
1)	$A = 1$	$B = 1$
2)	$x = B$	$y = A$

Figure 2: Instruction reordering example [4]

Under sequential consistency, when both processes have terminated, the values assigned to x and y at line 2 cannot both be the initial values. At least one of x or y must be assigned the value 1. A CPU, however might reorder the instructions at line 2 before the writes on line 1 because they are to different locations, and in a uniprocessor environment such a reordering could not change the program outcome. In that case both x and y could be assigned the values 0, thus violating sequential consistency.

Our framework (Section 2) and our total-store order machine running example (Section 3) allow for such possible reorderings that are due to components

outside of the shared memory system. So, if we are provided with the possible reorderings due to components such as CPUs or compilers, and with the memory system architecture, we can construct a programmer-centric specification of what computations can arise when a given programs runs on that system.

To illustrate the impact of the combination of a total-store-order memory system combined with some CUP reorderings, Section 4 revisits Peterson's mutual exclusion algorithm before making some concluding observations.

Our work on write-buffer architectures originally appeared in [5], which assumed that processors issue instructions in the order they appear in their programs. In this paper, we relax this assumption, capturing compiler and processor instruction-reordering optimizations. Our model and proofs are revised and our proofs are presented in a new diagrammatic notation, which we introduced in a paper that is under review [3] and is concerned with memory consistency models arising from distributed-shared memory in message-passing systems.

2 Framework for specifying memory consistency

We say that a multiprocess system M has *weak memory consistency* if there are programs for M that can produce executions that are not sequentially consistent. A memory consistency model for M provides a way for the programmer to determine what executions of a given multiprocess program can arise when it is run on M.

This section briefly presents our framework for specifying memory consistency models and a general methodology for establishing that an abstract model correctly captures a concrete architecture. Our framework has been developed and used in previous work [5, 7, 8, 3]. It also benefited from numerous previous papers in the area of memory consistency modelling (for example, [1, 2, 10, 12]).

2.1 Specifying weak memory consistency systems

The literature contains many descriptions of various memory consistency models. These range from informal English descriptions to formal mathematical ones. Steinke and Nutt [18] provide references to several memory consistency models and provide a unified way to define them. Our framework is close to theirs.

Frameworks for specifying concurrent systems are diverse — the most common are based on process algebras and automata theory. Communicating Sequential Processes (CSP) [9] and the Calculus of Communicating Systems (CCS) [15] are classic examples of process algebras. They start with a basic set of processes, then combine them into larger systems using various algebraic operators. In the Input-Output Automata (IOA) language [14], processes are specified as automata that communicate by action synchronization. Actions in IOA are automata

transitions that can arbitrarily modify local state. They are specified in an imperative language that is essentially pseudocode, making IOA especially useful for reasoning about algorithms written in imperative programming languages. The Temporal Logic of Actions, TLA [13], specifies automata using a mathematical language. It avoids programming languages, making it attractive to hardware designers. TLA also provides many tools for reasoning about automata in general, and can be used with IOA [16].

Framework:

We model a multiprocess system as a collection of shared objects V, a collection of processes P that access private objects and the objects in V, and a memory consistency model that defines exactly what computations can arise when the processes in P all run asynchronously while operating on V. (Our framework is also applicable to message-passing systems, but this is beyond the scope of this paper.)

A shared object is defined by a *sequential specification* [10] which determines the set of *valid* sequences of operations for each such object. This can be done 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, or by directly defining the set of sequences. For example, shared variables, which are the most common objects in this paper are specified as follows. A *(shared) variable*, x, is the set of sequences over operations of the type $x.\textsc{write}(\cdot)$ and $x.\textsc{read}()$ that satisfy: The output value returned by each $x.\textsc{read}()$ operation is the same as the input value written by the most recent preceding $x.\textsc{write}(\cdot)$ operation in the sequence, if such an $x.\textsc{write}(\cdot)$ exists, and is $\perp$ otherwise.

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 model an *individual (computer) process* as a sequential program, which generates a sequence of *operation invocations*. A *(completed) operation* is an operation invocation together with its response. The order of operations invocations that is dictated by the flow control of the program is called *program order*.

A *multiprogram* is a collection of individual processes. A *computation* of the multiprogram is formed by *arbitrarily* completing each operation invocation, in each individual process sequence of invocations, with a response. Thus, a computation is a collection of sequences of operations — one sequence, in program order, for each process in the multiprogram. Program order on operation invocations is naturally extended to define *program order* on the set of completed operations of a computation. We denote this unrestricted set of computations of a multiprogram P by $C(P)$. The subset of $C(P)$ that could actually result from the

execution of the multiprogram depends upon the system's architecture. A *memory consistency model* is a predicate defined on the set of all possible computations of a multiprogram; it filters these computations to include only those that could arise on the architecture being modeled. The subset of $C(P)$ that satisfies the memory consistency predicate, MC, is denoted $C(\mathsf{MC}, P)$.

We use the following notation, terminology and conventions for the remainder of this paper. For a computation C of a multiprogram P, O_C denotes all the operations of C. A completed operation OPER with input u that returns a value v is denoted $\frac{\text{OPER}(u)}{v}$. For a set of operations O, $O|_{\text{TYPE}}$ denotes the subset of all operations with type matching TYPE. For example, $O|_{\text{WRITE}}$ denotes the subset of all WRITE operations. The *program order relation on* O_C, denoted $\xrightarrow{\text{prog}_C}$, is the partial order formed by the union of the individual process program orders. For all these notations we omit the subscript C when it is obvious. Given a total order on a finite set, there is only one sequence of all the elements of the set that realizes that total order. Therefore, we sometimes overload the term *total order* for a finite set A: it refers to either the set of ordered pairs $(A, \overset{\text{T}}{\rightarrow})$ in the order, or the sequence, which we denote by T, that realizes that total order.

Using this framework, the predicate for sequential consistency, SC becomes:

Definition 2.1. $\mathsf{SC}[C] \overset{\text{def}}{=} \exists$ *valid total order* $(O_C, \overset{\text{L}}{\rightarrow})$ *satisfying* $(O_C, \xrightarrow{\text{prog}}) \subseteq (O_C, \overset{\text{L}}{\rightarrow})$.

In Section 3, we will see that there is a programmer-centric definition of the total-store-order machine that is a natural relaxation of sequential consistency as defined above.

2.2 Proving correctness of a memory consistency predicate

Given an architecture, called the *target* model, and a programmer-centric specification, called the *specified* model, for the memory consistency of that architecture, we need to prove that the specification is correct. That is, we need to prove that the abstract memory consistency predicate captures exactly the computations that can arise on the corresponding architecture. We now describe the general set-up for such proofs.

The transformation of the programmers' code to machine events converts each *specified* operation into a subroutine of operations on *target* components. The transformations of each instruction in the code, when concatenated in program order, induce a natural transformation of the entire specified multiprogram to a target multiprogram. To prove that the programmer-centric memory consistency predicate is correct, we must show the possible computations of these two multiprograms that can arise from their respective memory consistency models,

have the same "outcome". We make this precise as follows. Let $\tau(P)$ denote a transformation of multiprogram P. The possible computations of multiprogram P (respectively, $\tau(P)$) on the specified (respectively, target) memory consistent model MCModel (respectively, MCArch) is the set $C(P, \mathsf{MCModel})$ (respectively, $C(\tau(P), \mathsf{MCArch})$). But $\tau(P)$ transforms specified operation invocations that require a response into subroutines that return a response. So these returned responses can be used to *interpret* each computation in $C(\tau(P), \mathsf{MCArch})$ as a computation of P. We need to show that each such interpreted computation could have arisen in the specified model. That is, we must show that the interpretation of any computation in $C(\tau(P), \mathsf{MCArch})$ is in $C(P, \mathsf{MCModel})$. If this is satisfied for any P, we say that $\tau(\cdot)$ *correctly implements* MCModel on MCArch. Figure 3 depicts this proof obligation. If, in addition, for every computation $C \in C(P, \mathsf{MCModel})$, there

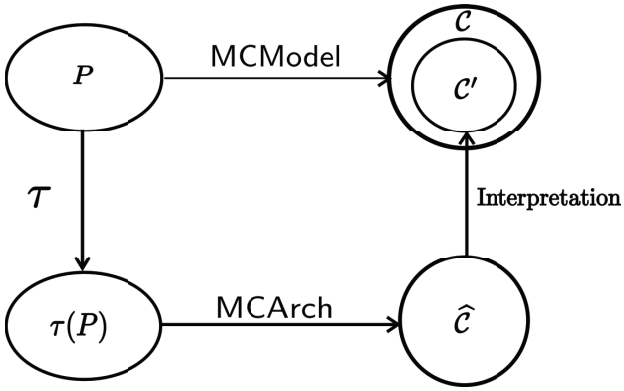


Figure 3: Proof obligation

is a computation $\widehat{C}$ in $C(\tau(P), \mathsf{MCArch})$ such that the interpretation of $\widehat{C}$ is C, then we say that $\tau(\cdot)$ *exactly implements* MCModel on MCArch. For clarity, SMALL CAPS font is used to denote specification level operations; `teletype` is used to denote target level operations. To emphasize that a component belongs to the target level, its name is sometimes annotated with a "hat" as in $\widehat{name}$.

The memory consistency definitions in Section 3 are relatively simple. They require that for every computation there is a single valid sequence of all its operations that preserve some subset of program order. Thus, they are relaxations of SC, which requires all of program order to be preserved. Such a sequence is called a *witness* that C satisfies the memory consistency predicate MC, and is informally referred to as a *witness sequence*. More formally, we use the predicate Witness$[S, C, \mathsf{MC}]$ to assert that the sequence S witnesses that C satisfies MC. For architectures other than the total-store-order machine used in this paper, the memory consistency predicate may require the existence of a collection of valid sequences each of which extends some partial order together with some agreement properties between these sequences. The natural extension of these definitions to the more general case then applies.

Let P be a specified program and $\tau(P)$ be a transformation of that program. The proofs are generally structured into three steps:
Assume: $\widehat{C} \in C(\tau(P), \mathsf{MCArch})$. Let $C \in C(P)$ be the interpretation of $\widehat{C}$.
Build: Choose a sequence $\widehat{S}$ such that $\mathsf{Witness}[\widehat{S}, \widehat{C}, \mathsf{MCArch}]$. Use $\widehat{S}$ to construct a corresponding sequence S for the operations in C.
Verify: Show that $\mathsf{Witness}[S, C, \mathsf{MCModel}]$.

The Verify step is typically long and requires several lemmas.

3 Memory Consistency of total-store-order systems

This section first constructs an operational specification of the total-store-order machine. Then we provide a programmer-centric specification and finally give the proof that the operational specification correctly implements the programmer-centric model.

This section uses work that appeared in an earlier paper [5], but differs in several ways. That paper modeled the total-store-order write-buffer machine only when the processes issue instructions in program order. Here we extend this to a model that is parametrized by an arbitrary partial order that reflects only what is guaranteed by the issue order. Thus the programmer-centric model of the total-store order machine captures the combined effects of CPU reordering and compiler optimizations together with the weakened guarantees due to the memory structure that uses write-buffers. In addition, the models and the proof in this section are revised and the diagrammatic form of the proofs is new.

3.1 Operational specification of a total-store-order system

The introduction of this paper includes an informal description of how the total-store-order write-buffer operates. In addition to the reordering caused by the write-buffers, CPU and compiler optimizations often reorder, overlap and pipeline operations, causing a process to invoke its operations in an order different from program order. We use our framework to formalize a total-store-order memory system that also allows such out of order invocations, by defining the objects, the processes and the memory consistency predicate of the system.

Let P be a multiprogram that operates on a set of shared variables V. To emphasize that, for this application of the framework, we are working at an assemble level on a SPARC-like multiprocessor, for the rest of this paper we use LOAD instead of READ and STORE instead of WRITE for the shared variable operations used by the programmer.

Objects The objects used by the total-store-order machine are write-buffers and shared memory represented as follows. For each variable x in V there is a corresponding main memory variable $\widehat{x}$ in a set called $\widehat{V}$. To emphasize that the operations are on the shared memory of the target total-store-order machine, READ and WRITE are renamed `mread` and `mwrite` respectively. For each processor p in P there is a corresponding list object $\widehat{L_p}$ that contains a list of ordered pairs of the form $(\widehat{x}, val)$ where $\widehat{x} \in \widehat{V}$ and val is a value. $\widehat{L_p}$ supports the three operations:

$\widehat{L_p}$.`append`$(\widehat{x},val)$ that appends the pair $(\widehat{x}, val)$ to $\widehat{L_p}$,

$\widehat{L_p}$.`delete`$(\widehat{x},val)$ that removes the pair $(\widehat{x}, val)$ from $\widehat{L_p}$, and

$\widehat{L_p}$.`last`$(\widehat{x})$ that returns the value component (i.e. second component) of the latest pair in $\widehat{L_p}$ that has $\widehat{x}$ as its first component, if such a pair exists, and $\perp$ otherwise.

The set of objects in the system is: $\widehat{V} \cup \bigcup_{p \in P} \{\widehat{L_p}\}$.

A sequence of `append`, `delete` and `last` operations on a list $\widehat{L}$ is valid if the value returned by each `last` is the one given by the operational definition above.

Processes The processes in our system arise from replacing each STORE and each LOAD operation invocation in P with the transformation τ defined in Algorithms 2 and 3 to yield the transformed multiprogram $\tau(P)$.

Algorithm 2: $\tau(x.\text{LOAD}_p())$

```
1 val ← L̂_p.last(x̂)
2 if val = ⊥ then val ← x̂.mread()
3 return val
```

Algorithm 3: $\tau(x.\text{STORE}_p(val))$

```
1 L̂_p.append(x̂,val)
2 L̂_p.delete(x̂,val)
3 x̂.mwrite(val)
```

Memory Consistency The invocation of LOAD and STORE instructions may be out of order due to compiler or CPU reorderings. This will cause a corresponding relaxation of the order of the `last` and `append` operations in the target system. In addition, the total-store-order write-buffer architecture may cause the `delete` and `mwrite` operations of any STORE to be delayed. In spite of reordering due to either cause, some ordering properties of the operations are ensured.

- When a LOAD or STORE is invoked, the corresponding τ(LOAD) or τ(STORE) operations will be executed in program order (though not necessarily contiguously).

- The `last` and `append` operations will execute in the order in which the corresponding LOAD and STORE operations are invoked.
- Each process' `mwrite` operations occur in the same order as its corresponding `append` operations because each buffer empties in FIFO order.
- When a LOAD is invoked, the invoking process completes all of τ(LOAD) before continuing with another invocation, because each LOAD is blocking.

We formulate our memory consistency predicate by formalizing these guarantees of the system. Let $(O,\overset{R}{\rightarrow})$ denote the subset of $(O,\overset{prog}{\longrightarrow})$ that is guaranteed to be maintained by the order in which LOAD and STORE instructions are invoked even after the reorderings due to compiler and/or CPU reorderings. Let $\widehat{O}$ be the set of all the operations of a computation $\widehat{C}$ of $\tau(P)$. Define $\texttt{parent}(\widehat{o_i})$ to be o if $\widehat{o_i}$ is in the program $\tau(o)$. Consider any two operations $\widehat{o_1}, \widehat{o_2} \in \widehat{O}$ and let $o_1, o_2 \in O$ such that $o_1 = \texttt{parent}(\widehat{o_1})$ and $o_2 = \texttt{parent}(\widehat{o_2})$. Define the following partial orders on $\widehat{O}$.

matching-ops: $(\widehat{o_1},\widehat{o_2}) \in (\widehat{O},\overset{match}{\longrightarrow})$ if and only if $o_1 = o_2$ and $\widehat{o_1}$ precedes $\widehat{o_2}$ in $\tau(o_1)$.

buffer-ops: $(\widehat{o_1},\widehat{o_2}) \in (\widehat{O},\overset{buff}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O,\overset{R}{\rightarrow})$ and $\widehat{o_1}, \widehat{o_2} \in \widehat{O}|_{\texttt{append}\cup\texttt{last}}$.

fifo-writes: $(\widehat{o_1},\widehat{o_2}) \in (\widehat{O},\overset{fifo}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O,\overset{R}{\rightarrow})$ and $\widehat{o_1},\widehat{o_2} \in \widehat{O}\,|_{\texttt{mwrite}}$.

blocking-loads: $(\widehat{o_1},\widehat{o_2}) \in (\widehat{O},\overset{block}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O,\overset{R}{\rightarrow})$ and $\widehat{o_1} \in \widehat{O}|_{\texttt{mread}\cup\texttt{last}}$.

tso-ops: $(\widehat{o_1},\widehat{o_2}) \in (\widehat{O},\overset{tso}{\longrightarrow})$ if and only if $(\widehat{o_1},\widehat{o_2}) \in (\widehat{O},\{\overset{match}{\longrightarrow} \cup \overset{buff}{\longrightarrow} \cup \overset{fifo}{\longrightarrow} \cup \overset{block}{\longrightarrow}\}^+)$.

We can now define the predicate that captures the computations of a total-store-order machine when the LOAD and STORE instructions are invoked in any order that extends the subset of program order given by $(O,\overset{R}{\rightarrow})$.

Definition 3.1. $\mathsf{TSOArch}(\overset{R}{\rightarrow})[\widehat{C}] \overset{def}{=} \exists$ *a valid total order* $(\widehat{O}_{\widehat{C}}, \overset{TSOArch}{\longrightarrow})$ *satisfying* $(\widehat{O}_{\widehat{C}}, \overset{tso}{\longrightarrow}) \subseteq (\widehat{O}_{\widehat{C}}, \overset{TSOArch}{\longrightarrow})$.

3.2 Programmer-centric specification of a system

Again we define a system by specifying its objects, processes and memory consistency predicate, but this time without reference to the write-buffer hardware and its operations.

Objects The objects are a set V of shared variables, and arbitrary private objects. Each shared variable supports the READ operation, denoted LOAD, and the WRITE operation, denoted STORE.

Processes Processes are programs that apply operations to their private objects and the shared variables in V.

Memory Consistency Let O be the set of all the operations of a computation C of P where the shared variables are V. A LOAD operation by process p is *domestic* if the value it returns was STOREed by p. Otherwise it is *foreign*. Define the following partial orders on O as a function of a partial order $(O,\overset{R}{\rightarrow})$, which is any fixed subset of $(O,\overset{prog}{\longrightarrow})$.

same-object: $(o_1,o_2) \in (O,\overset{s.o}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O\mid_x,\overset{R}{\rightarrow})$ for some $x \in V$.

preceding-foreign-load: $(o_1,o_2) \in (O,\overset{p.f.l}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O,\overset{R}{\rightarrow})$ and o_1 is a foreign load.

following-store: $(o_1,o_2) \in (O,\overset{f.s}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O,\overset{R}{\rightarrow})$ and $o_2 \in O|_{\text{STORE}}$.

TSO-abstract: $(o_1,o_2) \in (O,\overset{TSO}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O, \{\overset{s.o}{\longrightarrow} \cup \overset{p.f.l}{\longrightarrow} \cup \overset{f.s}{\longrightarrow}\}^+)$.

Define the abstract memory consistency predicate:

Definition 3.2. $\mathsf{TSOModel(R)}[C] \overset{\text{def}}{=} \exists$ *a valid total order* $(O_C, \overset{TSOModel}{\longrightarrow})$ *satisfying* $(O_C, \overset{TSO}{\longrightarrow}) \subseteq (O_C, \overset{TSOModel}{\longrightarrow})$.

3.3 Equivalence of the total-store-order architecture and our abstract model

In this subsection we prove that for any computation of $\tau(P)$ on the total-store-order write-buffer architecture parametrized by any partial order subset of program order (that is, one that satisfies predicate $\mathsf{TSOArch}(\overset{R}{\rightarrow})$), its interpretation as a computation of P satisfies $\mathsf{TSOModel}(\overset{R}{\rightarrow})$).

Theorem 3.3. *Algorithms 2 and 3 together correctly implement* $\mathsf{TSOModel}(\overset{R}{\rightarrow})$ *on* $\mathsf{TSOArch}(\overset{R}{\rightarrow})$ *for any* $(O, \overset{R}{\rightarrow}) \subseteq (O, \overset{prog}{\longrightarrow})$.

Proof. Assume-TSO: Suppose $\widehat{C} \in C(\tau(P), \mathsf{TSOArch}(\xrightarrow{R}))$. Let $C \in C(P)$ be the interpretation of $\widehat{C}$.

Build-TSO: Choose a sequence $\widehat{S}$ such that $\mathsf{Witness}[\widehat{S}, \widehat{C}, \mathsf{TSOArch}(\xrightarrow{R})]$. Construct S from $\widehat{S}$ as follows:

$\widehat{S_a}$: Discard: Discard all $\frac{\texttt{last}}{\perp}$, `append` and `delete` operations.

$\widehat{S_b}$: Reorder: For every `mwrite` operation $\hat{o}$, move all the $\frac{\texttt{last}}{v}$ operations that return the value written by $\hat{o}$, to immediately after $\hat{o}$ while maintaining the relative order that these $\frac{\texttt{last}}{v}$ operations had in $\widehat{S}$.

S: Lift: Replace every $x.\texttt{mwrite}(v)$ with a $x.\textsc{store}(v)$. Replace every $\frac{x.\texttt{mread}()}{v}$ and $\frac{x.\texttt{last}()}{v}$ by a $\frac{x.\textsc{load}()}{v}$.

Verify-TSO: We need to show that $\mathsf{Witness}[S, C, \mathsf{TSOModel}(\xrightarrow{R})]$. From the construction it is clear that S contains exactly the operations in O_C. From the definition of $\mathsf{TSOModel}(\xrightarrow{R})$, the proof is completed as follows:

Proof Obligation	Lemma
S is valid.	3.4
S extends $(O_C, \xrightarrow{\text{p.f.l}})$.	3.5
S extends $(O_C, \xrightarrow{\text{f.s}})$.	3.6
S extends $(O_C, \xrightarrow{\text{s.o}})$.	3.7

□

Both $\tau(x.\textsc{load}_p())$ (Algorithm 2) and $\tau(x.\textsc{store}_p(val))$ (Algorithm 3) invoke operations on both the list L_p of process p, and (possibly, in the case of LOAD) on a shared memory variable. In the following proofs, it is convenient to use the term `buffer-op()` (respectively, `memory-op()`) to refer to the operation on the list object (respectively, the shared variable) in either $\tau(x.\textsc{load}_p())$ or $\tau(x.\textsc{store}_p(val))$. Also, we distinguish the case that $\tau(x.\textsc{load}_p())$ returns at line 1 or line 2 of Algorithm 2 by refering to the first as a BUFFER-LOAD and the second as a MEMORY-LOAD.

Lemma 3.4. *The sequence S as defined in the Build-TSO step of Theorem 3.3 is valid.*

Proof. Since $\widehat{S}$ is valid, and no `mwrite` or `mread` is moved in the construction of $\widehat{S_b}$ from $\widehat{S}$, all `memory-op()`s are valid in $\widehat{S_b}$. In the Lift step, each `mwrite` is renamed STORE and each `mread` is renamed LOAD without changing any parameters, so these remain valid in S. In the Reorder step of Build-TSO each `last` that returned a non-$\perp$ value is moved to immediately after the `mwrite` operation whose value it returns. So, when renamed as a LOAD operation, it becomes valid in S. □

Lemma 3.5. *The sequence S as defined in the Build-TSO step of Theorem 3.3 satisfies: S extends $(O, \overset{\text{p.f.l}}{\longrightarrow})$.*

Proof. Consider arbitrary operations o_1, o_2 where $(o_1, o_2) \in (O, \overset{\text{p.f.l}}{\longrightarrow})$. By definition of the preceding foreign load partial order, if $(o_1, o_2) \in (O, \overset{\text{p.f.l}}{\longrightarrow})$ then $(o_1, o_2) \in (O, \overset{\text{R}}{\longrightarrow})$.

Case 1 - o_2 is a MEMORY-LOAD or a STORE operation:

Thus, $\text{LOAD}_p^1 \xrightarrow{\text{R}} o_2$

$\overset{\tau}{\Longrightarrow}$ $\underset{\bot}{\texttt{last}_p^1} \xrightarrow{\text{match}} \texttt{mread}_p^1 \xrightarrow{\text{block}} \texttt{buffer-op}(o_2) \xrightarrow{\text{match}} \texttt{memory-op}(o_2)$

$\overset{\text{TSOArch}}{\Longrightarrow}$ $\texttt{mread}_p^1 \xrightarrow{\widehat{S}} \texttt{buffer-op}(o_2) \xrightarrow{\widehat{S}} \texttt{memory-op}(o_2)$

$\overset{\text{Build-TSO}}{\Longrightarrow}$ $\text{LOAD}_p^1 \xrightarrow{\text{S}} o_2$

Case 2 - o_2 is a BUFFER-LOAD:

$\Longrightarrow$ $\text{LOAD}_p^1 \xrightarrow{\text{R}} \text{LOAD}_p^2$

$\overset{\tau}{\Longrightarrow}$ $\underset{\bot}{\texttt{last}_p^1} \xrightarrow{\text{match}} \texttt{mread}_p^1 \xrightarrow{\text{block}} \underset{v}{\texttt{last}_p^2}$

Let the STORE operation that writes the value returned by LOAD_p^2's `last` operation be STORE_p^*. Let $(\widehat{O}, \overset{\text{val}}{\longrightarrow})$ denote the partial order that dictates such orderings which are crucial for the validity of the transformed program. Naturally, if $\widehat{o_1} \overset{\text{val}}{\longrightarrow} \widehat{o_2}$ then $\widehat{o_1} \overset{\widehat{S}}{\rightarrow} \widehat{o_2}$.

$\overset{\tau}{\Longrightarrow}$ $\underset{\bot}{\texttt{last}_p^1} \xrightarrow{\text{match}} \texttt{mread}_p^1 \xrightarrow{\text{block}} \underset{v}{\texttt{last}_p^2}$

$\texttt{append}_p^* \xrightarrow{\text{match}} \texttt{delete}_p^* \xrightarrow{\text{match}} \texttt{mwrite}_p^*$, with $\texttt{append}_p^* \xrightarrow{\text{val}} \underset{v}{\texttt{last}_p^2}$ and $\underset{v}{\texttt{last}_p^2} \xrightarrow{\text{val}} \texttt{delete}_p^*$

$\overset{\text{TSOArch}}{\Longrightarrow}$ $\texttt{mread}_p^1 \xrightarrow{\widehat{S}} \underset{v}{\texttt{last}_p^2} \xrightarrow{\widehat{S}} \texttt{mwrite}_p^*$

$\overset{\text{Build-TSO}}{\Longrightarrow}$ $\texttt{mread}_p^1 \xrightarrow{\widehat{S}_b} \texttt{mwrite}_p^* \xrightarrow{\widehat{S}_b} \underset{v}{\texttt{last}_p^2}$

$\overset{\text{Build-TSO}}{\Longrightarrow}$ $\text{LOAD}_p^1 \xrightarrow{\text{S}} \text{LOAD}_p^2$

□

Lemma 3.6. *The sequence S as constructed as defined in the Build-TSO step of Theorem 3.3 satisfies: S extends $(O, \overset{\text{f.s}}{\longrightarrow})$.*

Proof. Consider arbitrary operations o_1, o_2 where $(o_1, o_2) \in (O, \overset{\text{f.s}}{\longrightarrow})$. By definition of the following store partial order, if $(o_1, o_2) \in (O, \overset{\text{f.s}}{\longrightarrow})$ then $(o_1, o_2) \in (O, \overset{\text{R}}{\rightarrow})$.

Case 1 - o_1 is a MEMORY-LOAD: Since a MEMORY-LOAD operation behaves exacltly like a foreign LOAD operation, the proof from Lemma 3.5 applies, and it follows that $o_1 \overset{\text{S}}{\rightarrow} o_2$.

Case 2 - o_1 is a BUFFER-LOAD:

$$\Longrightarrow \quad \text{LOAD}_p^1 \xrightarrow{\text{R}} \text{STORE}_p^2$$

$$\overset{\tau}{\Longrightarrow} \quad \frac{\texttt{last}_p^1}{v} \xrightarrow{\text{block}} \texttt{append}_p^2 \xrightarrow{\text{match}} \texttt{mwrite}_p^2$$

Let the STORE operation that writes the value returned by LOAD_p^1's `last` operation be STORE_p^*. So then, STORE_p^* is the most recent store to object x before LOAD_p^1 and $\text{STORE}_p^* \overset{\text{R}}{\rightarrow} \text{LOAD}_p^1 \overset{\text{R}}{\rightarrow} \text{STORE}_p^2$ holds.

$$\overset{\tau}{\Longrightarrow} \quad \frac{\texttt{last}_p^1}{v} \xrightarrow{\text{block}} \texttt{append}_p^2 \xrightarrow{\text{match}} \texttt{mwrite}_p^2$$

$$\texttt{append}_p^* \xrightarrow{\text{match}} \texttt{delete}_p^* \xrightarrow{\text{match}} \texttt{mwrite}_p^* \quad \text{(val: append}_p^* \to \frac{\texttt{last}_p^1}{v}\text{; val: delete}_p^* \to v\text{; fifo: mwrite}_p^* \to \texttt{mwrite}_p^2\text{)}$$

$$\overset{\text{TSOArch}}{\Longrightarrow} \quad \frac{\texttt{last}_p^1}{v} \xrightarrow{\widehat{S}} \texttt{mwrite}_p^* \xrightarrow{\widehat{S}} \texttt{mwrite}_p^2$$

$$\overset{\text{Build-TSO}}{\Longrightarrow} \quad \texttt{mwrite}_p^* \xrightarrow{\widehat{S}_b} \frac{\texttt{last}_p^1}{v} \xrightarrow{\widehat{S}_b} \texttt{mwrite}_p^2$$

$$\overset{\text{Build-TSO}}{\Longrightarrow} \quad \text{LOAD}_p^1 \xrightarrow{\text{S}} \text{STORE}_p^2$$

Case 3 - o_1 is a STORE:

Thus, $\quad \text{STORE}_p^1 \xrightarrow{\text{R}} \text{STORE}_p^2$ (fifo)

$$\overset{\tau}{\Longrightarrow} \quad \texttt{append}_i^1 \xrightarrow{\text{match}} \texttt{mwrite}_i^1 \qquad \texttt{append}_i^2 \xrightarrow{\text{match}} \texttt{mwrite}_i^2$$

$$\overset{\text{TSOArch}}{\Longrightarrow} \quad \texttt{mwrite}_p^1 \xrightarrow{\widehat{S}} \texttt{mwrite}_p^2$$

$$\overset{\text{Build-TSO}}{\Longrightarrow} \quad \text{STORE}_p^1 \xrightarrow{\text{S}} \text{STORE}_p^2$$

□

Lemma 3.7. *The sequence S as defined in the Build-TSO step of Theorem 3.3 satisfies: S extends $(O, \overset{\text{s.o}}{\longrightarrow})$.*

Proof. Consider arbitrary operations o_1, o_2 where $(o_1, o_2) \in (O, \overset{\text{s.o}}{\longrightarrow})$. By definition of the same object partial order, if $(o_1, o_2) \in (O, \overset{\text{s.o}}{\longrightarrow})$ then $(o_1, o_2) \in (O, \overset{\text{R}}{\rightarrow})$.

Let x be the object that operations o_1 and o_2 are acting upon.

Case 1 - o_1 is a MEMORY-LOAD: Since a MEMORY-LOAD operation behaves exacltly like a foreign LOAD operation, from Lemma 3.5, it follows that $o_1 \xrightarrow{S} o_2$.

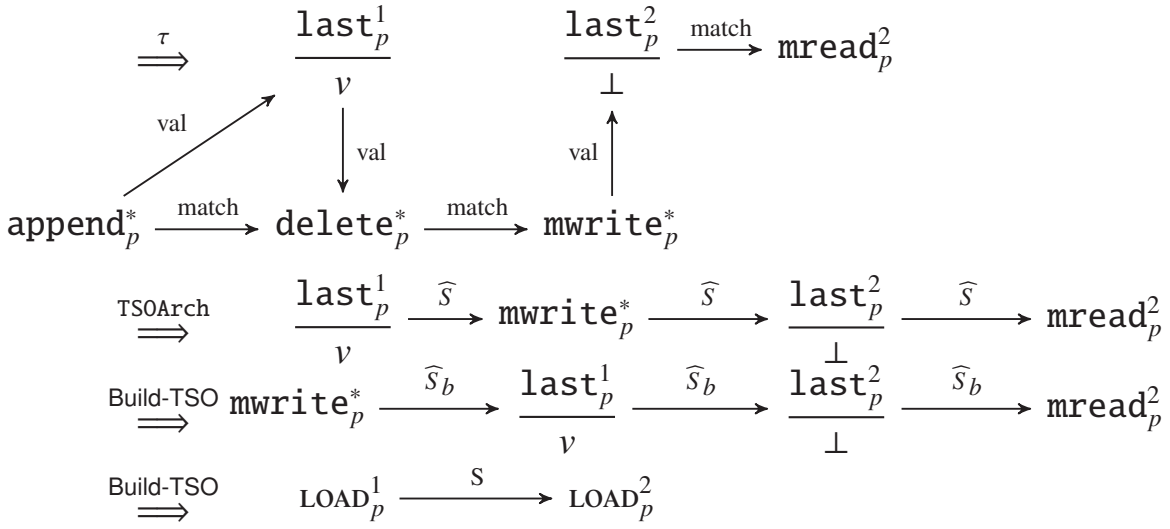
Case 2 - o_2 is a STORE: Then, from Lemma 3.6, it follows that $o_1 \xrightarrow{S} o_2$.

Case 3 - o_1 is a BUFFER-LOAD and o_2 is a MEMORY-LOAD:

$$\Longrightarrow \quad \text{LOAD}_p^1 \xrightarrow{R} \text{LOAD}_p^2$$

$$\overset{\tau}{\Longrightarrow} \quad \frac{\texttt{last}_p^1}{v} \xrightarrow{\text{block}} \frac{\texttt{last}_p^2}{\bot} \xrightarrow{\text{match}} \texttt{mread}_p^2$$

Let the STORE operation that writes the value returned by LOAD_p^1's `last` operation be STORE_p^*. So then, STORE_p^* is the most recent store to object x before LOAD_p^1 and $\text{STORE}_p^* \xrightarrow{R} \text{LOAD}_p^1 \xrightarrow{R} \text{LOAD}_p^2$ holds. Since STORE_p^* and LOAD_p^2 are both acting on the same object and LOAD_p^2's `last` operation returned $\bot$, it follows that the o_1's STORE_p^* operation is flushed to main memory before LOAD_p^2's `last` operation.

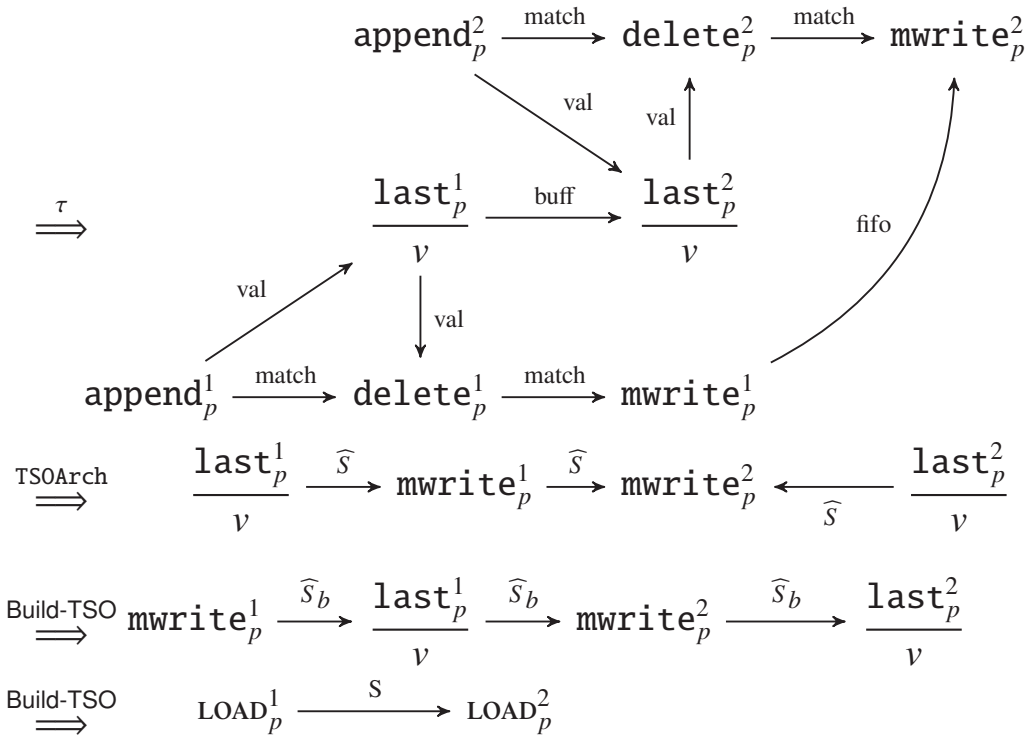


Case 4 - o_1,o_2 are both BUFFER-LOAD:

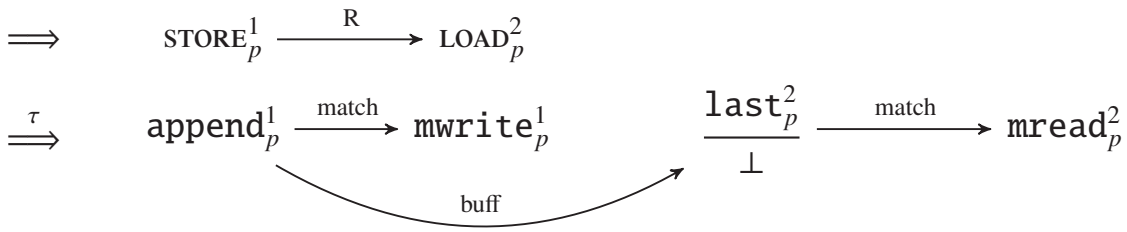
$$\Longrightarrow \quad \text{LOAD}_p^1 \xrightarrow{R} \text{LOAD}_p^2$$

$$\overset{\tau}{\Longrightarrow} \quad \frac{\texttt{last}_p^1}{v} \xrightarrow{\text{buff}} \frac{\texttt{last}_p^2}{v}$$

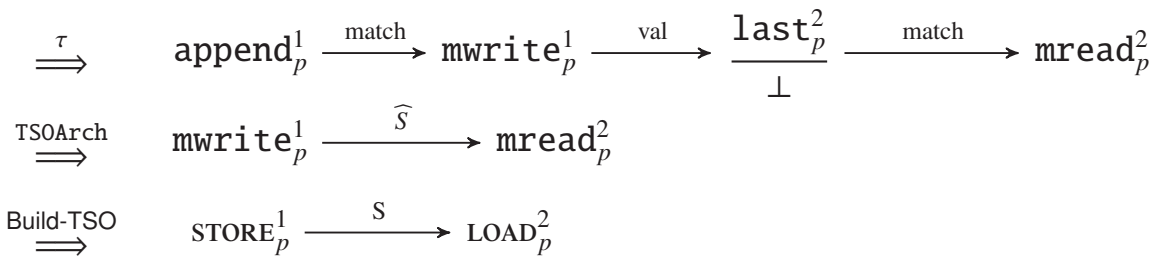
Let STORE_p^1 and STORE_p^2 be the STORE operations that writes the value returned by LOAD_p^1 and LOAD_p^2's `last` operation, respectively. So then, STORE_p^1 and STORE_p^2 are the most recent stores to object x before LOAD_p^1 and LOAD_p^2, respectively. Clearly, $\text{STORE}_p^1 \xrightarrow{R} \text{STORE}_p^2$ holds.



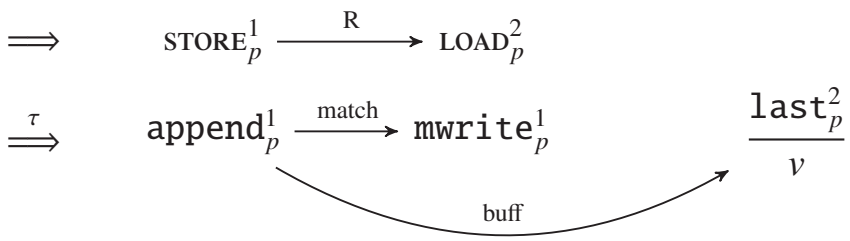
Case 5 - o_1 is a STORE operation and o_2 is a MEMORY-LOAD:



Since o_1 and o_2 are both acting on the same object and o_2's `last` operation returned $\perp$, it follows that the o_1's `mwrite` operation is flushed to main memory

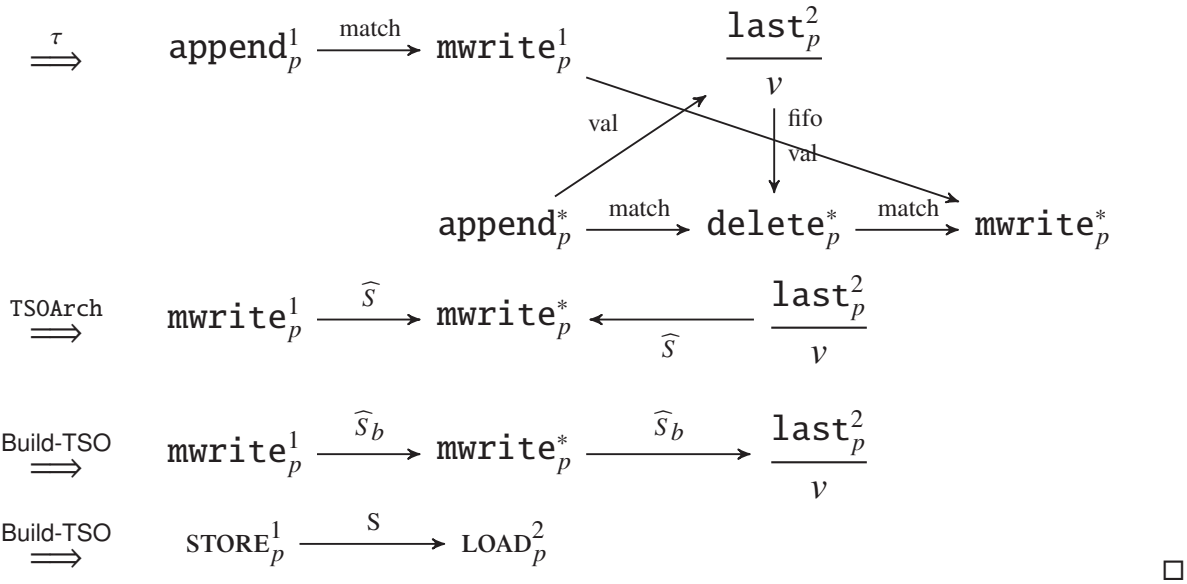


Case 6 - o_1 is a STORE operation and o_2 is a BUFFER-LOAD:



Let x be the object that operations o_1 and o_2 are acting upon. Let the STORE operation that writes the value returned by o_2's `last` operation be o_*. So then, if

$o_* \neq o_1$ then o_* is the most recent store to object x before o_2 and $o_1 \xrightarrow{R} o_*$ holds.



□

We do not present the other direction: that every computation C that satisfies TSOModel($\xrightarrow{R}$) is the interpretation of a computation that satisfies TSOArch($\xrightarrow{R}$). In fact this also holds. The proof of this for the case when $(O, \xrightarrow{R}) = (O, \xrightarrow{prog})$ has been previously published [5]. The extension to $(O, \xrightarrow{R}) \subseteq (O, \xrightarrow{prog})$ is straightforward and similar to the extension in the other direction in this paper. Combining both gives the theorem:

Theorem 3.8. *Algorithms 2 and 3 together exactly implement* TSOModel($\xrightarrow{R}$) *on* TSOArch($\xrightarrow{R}$) *for any* $(O, \xrightarrow{R}) \subseteq (O, \xrightarrow{prog})$.

4 Impact of relaxing program order

The partial order $(O, \xrightarrow{R})$ used as a parameter in the predicate TSOModel($\xrightarrow{R}$) derived in Section3, can be an arbitrary subset of program order. But because it is meant to capture the subset of $(O, \xrightarrow{prog})$ that is guaranteed by a real machine, processor, or compiler, there are some natural instantiations of $(O, \xrightarrow{R})$ that are worth defining and investigating further.

The CPU and compiler optimizations that weaken $(O, \xrightarrow{prog})$ to $(O, \xrightarrow{R})$ must, nevertheless, ensure that when a process runs by itself, the responses returned by the actual order of invocations are the same as responses that would be returned when the process executes sequentially invoking one operation after another in program order. This is called *processor-self-consistency*, meaning that a process may execute out of program order but all control dependencies and location dependencies are honored. Processor-self-consistency seems to be the minimum

guarantee of what arises after reordering due to compilers or CPUs since otherwise even the computation of a single thread in isolation would be too unpredictable. Define the partial orders:

control-dependence: $(o_1,o_2) \in (O,\overset{\text{c.d}}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O,\overset{\text{prog}}{\longrightarrow})$ and execution of o_2 is control dependent on o_1.

processor-self-consistency-order: $(o_1,o_2) \in (O,\overset{\text{psc}}{\longrightarrow})$ if and only if $(o_1,o_2) \in (O, \{\overset{\text{c.d}}{\longrightarrow} \cup \overset{\text{s.o}}{\longrightarrow}\}^+)$.

At the other extreme, program order seems to be the strongest possible guarantee in any purely asynchronous system.

Intermediate ordering guarantees can be imposed by barrier synchronization instructions, as shown in Figure 4. When inserted into a program, a barrier further restricts the order in which LOAD and STORE instructions are completed. The SPARC version 9 architecture introduced four basic barriers and its instruction set includes all combinations. An a-b-barrier, where $a, b \in \{\text{LOAD}, \text{STORE}\}$, ensures that

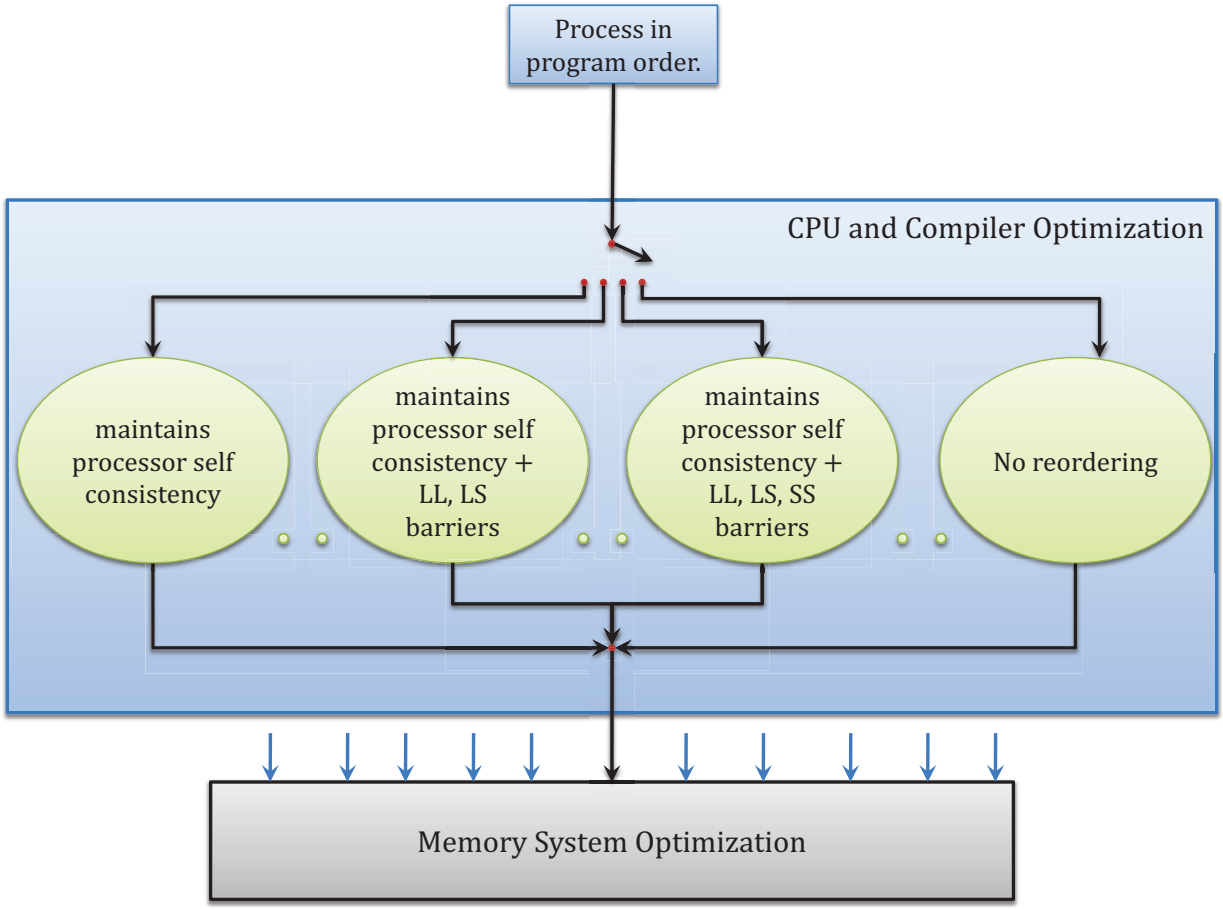


Figure 4: Issue order variants

for any operations o_1 of type a and o_2 of type b, if $o_1 \xrightarrow{\text{prog}} a\text{-}b\text{-barrier} \xrightarrow{\text{prog}} o_2$, then o_1 precedes o_2 in the total order guaranteed by TSOMODEL($\xrightarrow{R}$), for any partial order $\xrightarrow{R}$.

To illustrate the difference between TSOMODEL($\xrightarrow{\text{prog}}$) and TSOMODEL($\xrightarrow{\text{psc}}$), we return to Peterson's mutual exclusion algorithm (Algorithm 1) in Section 1, where we informally demonstrated how this algorithm fails for TSOMODEL($\xrightarrow{\text{prog}}$). Thus, it also fails for TSOMODEL($\xrightarrow{\text{psc}}$).

Barrier instructions are added to ensure the correctness of this algorithm for TSOMODEL($\xrightarrow{\text{prog}}$) and TSOMODEL($\xrightarrow{\text{psc}}$). To extend the definition of TSOMODEL($\xrightarrow{R}$) to include barriers, we revise the definition of a foreign LOAD. A LOAD r of some object x is *foreign* if r is a not domestic or if there is a STORE-LOAD-barrier that intervenes in program order between r and the most recent STORE to x that precedes r. The proof for this revision is beyond the scope of this paper. The intuition, however, is that a STORE-LOAD-barrier forces the pending STORES in the buffer to be committed to main memory before subsequent LOADS are issued. Hence, such LOAD operations return values from main memory.

To ensure the algorithm's correctness for TSOMODEL($\xrightarrow{\text{prog}}$), we now argue that it is necessary and sufficient to add a STORE-LOAD-barrier instruction between lines 2 and 3. In the following discussion, *before*, *after*, *precedes*, and *follows* are used in the context of the total orders guaranteed by TSOMODEL($\xrightarrow{\text{prog}}$) or TSOMODEL($\xrightarrow{\text{psc}}$).

Both STORE operations of lines 1 and 2 occur before the LOAD operation of line 3, due to the introduced barrier. The LOAD of flag in the while loop (line 3) is foreign since flag[$\bar{\iota}$] is only STOREed by $\bar{\iota}$ and LOADed by ι. Without the barrier, the LOAD of turn could be domestic but it becomes foreign after the insertion of the barrier between lines 2 and 3. So, both LOAD operations of the while loop are foreign. Hence, they precede any operations of the critical section in the total order guaranteed by TSOMODEL($\xrightarrow{\text{prog}}$). This observation also holds in TSOMODEL($\xrightarrow{\text{psc}}$).

The STORE in the exit section (line 4) should follow all the operations in the critical section. By the definition of TSOMODEL($\xrightarrow{\text{prog}}$), this is ensured for all STORE and foreign LOAD operations. This may not be the case for domestic LOADS in the critical section. However, since these are domestic, they will be returning values written by the same process and delaying them to follow the STORE of flag at line 4 does not affect correctness.

While this barrier is necessary and sufficient for this algorithm to be correct in TSOMODEL($\xrightarrow{\text{prog}}$), it is insufficient for TSOMODEL($\xrightarrow{\text{psc}}$). A STORE in the critical section could follow the STORE to flag in the exit section. These two STORE are related by program order, but they need not be related by processor self-consistency.

Hence, a STORE-STORE-barrier is also needed after the critical section and before line 4 to ensure the correctness of this algorithm in TSOMODEL($\overset{psc}{\longrightarrow}$).

References

[1] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the 5th International Symposium on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.

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

[3] Steven Cheng, Lisa Higam, and Jalal Kawash. Partition consistency: A case study in modeling systems with weak memory consistency and proving correctness of their implementations. under review.

[4] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[5] Lisa Higham, LillAnne Jackson, and Jalal Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM Trans. Comput. Syst.*, 25(1), 2007.

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

[7] Lisa Higham and Jalal Kawash. Tight bounds for critical sections in processor consistent platforms. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1072–1083, 2006.

[8] Lisa Higham and Jalal Kawash. Implementing sequentially consistent programs on processor consistent platforms. *J. Parallel Distrib. Comput.*, 68(4):488–500, 2008.

[9] C.A.R. Hoare. *Communicating Sequential Processes (Prentice-Hall International Series in Computer Science)*. Prentice Hall, April 1985.

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

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

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

[13] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, July 2002.

[14] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[15] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[16] Olaf Müller. I/o automata and beyond: Temporal logic and abstraction in isabelle. In Jim Grundy and Malcolm C. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 331–348. Springer, September 1998.

[17] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[18] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

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

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