Effective Program Verification for Relaxed Memory Models

January 31, 2008

Technical Report
MSR-TR-2008-12

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Effective Program Verification for Relaxed Memory Models

Sebastian Burckhardt   Madanlal Musuvathi

Microsoft Research

{sburckha,madanm}@microsoft.com

Abstract

Program verification for relaxed memory models is hard. The high degree of nondeterminism in such models challenges standard verification techniques. This paper proposes a new verification technique for the most common relaxation, store buffers. Crucial to this technique is the observation that all programmers, including those who use low-lock techniques for performance, expect their programs to be sequentially consistent. We first present a monitor algorithm that can detect the presence of program executions that are not sequentially consistent due to store buffers while *only* exploring sequentially consistent executions. Then, we combine this monitor with a stateless model checker that verifies that every sequentially consistent execution is correct. We have implemented this algorithm in a prototype tool called Sober and present experiments that demonstrate the precision and scalability of our method. We find relaxed memory model bugs in several programs, including a previously unknown bug in a production-level concurrency library that would have been difficult to find by other means.

1. Introduction

Developers of performance-critical multi-threaded software often try to avoid the overhead of traditional locking by either making direct use of hardware primitives for atomic operations (such as interlocked exchange, or compare-and-swap), or by employing regular loads and stores for synchronization purposes. Unfortunately, such "low-lock" programs are notoriously hard to get right [4, 21]. Subtle bugs can arise in these programs due to memory reordering caused by the relaxed memory model of the underlying hardware [1] . These errors are hard to find and debug as they most often show up only in specific thread interleavings and in particular hardware configurations. On the other hand, low-lock code is heavily used both in low-level libraries and in critical paths of a system. Because these parts are crucial to the reliability of the entire system, it is important to develop verification techniques.

In general, the same program may exhibit more executions on a relaxed model than on a sequentially consistent (SC) machine [19], as illustrated in Fig. 1. Let $\mathcal{T}_\pi^Y$ denote the set of executions of program π on memory model Y. Most existing program verification tools can not verify directly whether the executions in $\mathcal{T}_\pi^Y$ are correct (unless $Y = SC$). A few specialized memory model sensitive verification tools exist [4, 14, 23, 26, 27] but scalability and automation remain a challenge.

A key observation of this paper is that programmers, even those writing low-lock code, *expect* their programs to be sequentially consistent. They design their programs to be correct for SC executions and insert memory ordering fences to counter relaxations where necessary. In particular, any program execution that is not SC is almost always an error, resulting either from an insufficient use of fences or a misunderstanding of the underlying memory model.

This observation suggests that we can sensibly verify the relaxed executions $\mathcal{T}_\pi^Y$ by solving the following two verification problems separately:

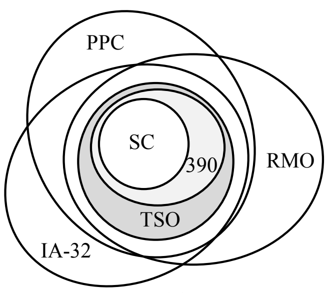
1. Use standard verification methodology for concurrent programs to show that the executions in $\mathcal{T}_\pi^{SC}$ are correct.
2. Use specialized methodology for *memory model safety* verification, showing that $\mathcal{T}_\pi^Y = \mathcal{T}_\pi^{SC}$. We say the program π is Y*-safe* if $\mathcal{T}_\pi^Y = \mathcal{T}_\pi^{SC}$.

In this paper we focus on verifying memory model safety for the most common relaxation in modern multiprocessors, *store buffers* with store-load forwarding. The corresponding memory model is historically called *TSO* (total store order) [25], and we use the terms *TSO*-safety and store buffer safety interchangeably. Under *TSO*, processors may delay the effect of a store instruction in a processor-local FIFO buffer (to hide the memory latency). While the values of these store instructions are immediately visible to the local processor, other processors see these values only when the store buffer is committed at a later time. Fig. 1(b) shows a simple example. We provide a rigorous characterization of *TSO* in Section 3.3.

Apart from the fact that store buffers are so common (as apparent in Fig. 1(a), $\mathcal{T}_\pi^{TSO} \subseteq \mathcal{T}_\pi^Y$ for almost all models Y), our motivation for focusing on *TSO* largely arises from the need to prepare the huge body of legacy code heavily optimized to run on x86 machines for future multicore chip generations. These processors are likely to make increased use of store buffers but are otherwise fairly conservative as far as the memory model is concerned [16].

The main contribution of this paper is a technique for checking the store buffer safety of a program while *only* exploring its sequentially consistent executions, which lets us perform the steps 1 and 2 above *simultaneously*. Our technique relies on a notion of *borderline* execution, which is an SC execution that can be extended into an execution in $\mathcal{T}_\pi^{TSO} \setminus \mathcal{T}_\pi^{SC}$. We establish that a program is store buffer safe *exactly* if there are no borderline executions (Theorem 7). Then we present an efficient, precise monitor for detecting borderline executions, using a novel *generalized* vector clock algorithm.

We have implemented these ideas in a prototype tool called Sober. Sober combines our store buffer safety monitor with the stateless model checker CHESS [22] which systematically enumerates the *SC* executions of a bounded concurrent test program and checks them for errors such as null pointers or assertion violations. In principle, Sober terminates with one of three possible outputs. First, Sober may detect a regular program error and output an erroneous execution. Second, Sober may report that the program is not store buffer safe. Finally, Sober may terminate without finding an error, proving that all *TSO* executions of the program are correct. In practice, exhaustive verification is too time-consuming for most



(a)

Initially: $X = Y = 0$

processor 1	processor 2
X = 1	Y = 1
r_1 = Y	r_2 = X

Eventually: $r_1 = 0, r_2 = 0$

When the two threads run on different processors, the stores to X and Y in the first line can possibly be delayed by the store buffers in these processors. Subsequent loads in the second line see the initial values of X and Y if the store buffers have not yet been committed.

(b)

Figure 1. (a) A comparison of various memory models [5, 9, 15, 17, 25]. (b) An execution that is possible on *TSO* but not on *SC*.

programs and we resort to iterative context-bounding [22], which provides verification guarantees up to a specific preemption bound.

Section 5 describes our initial experiments. Using Sober we found and fixed store buffer issues in several programs, including Dekker's mutual exclusion protocol [2] and the Bakery protocol [18]. We got our greatest success so far when we applied Sober to a component of a concurrency library at Microsoft. This component implements a low-lock datastructure. Sober demonstrated a store buffer problem that the developer immediately agreed was a real error. This bug was never detected during the extensive code-review and testing the component underwent.

2. Related Work

Prior work has addressed the verification of programs for relaxed memory models using explicit state enumeration [23, 6, 14] and using constraint solving [12, 28, 3, 4]. Our work improves upon them in scalability.

To our knowledge, this paper is the first to demonstrate the possibility of program verification without exploring the additional nondeterminism of memory-model relaxation. See the experiments in Huynh and Roychoudhury [14] for the state space explosion caused by this nondeterminism even for simple programs.

This paper is definitely not the first to observe that sequential consistency is the most natural memory model for programmers [19, 1, 13]. The Java Memory Model [20] guarantees sequential consistency for a broad class of programs, namely those which are data-race free. In contrast, our characterization of memory model safety *precisely* captures those programs which behave sequentially consistent in a memory model. In particular, a program with data-races might still be memory-model safe.

Specialized algorithms to automatically insert fences based on static analysis [24, 7] can guarantee memory-safety in principle. However, doubts remain about their precision in the presence of aliasing, loops, and conditionals and the performance implication of conservative fence insertion. Also, the memory models considered in these algorithms assume atomic memory and cannot model store buffers, the main emphasis of this paper.

3. Problem Formulation

We represent the relevant aspects of a program executions by a *memory trace*, or just trace. A trace is a collection of events, each representing a memory access (either a store, a load, or an interlocked operation[1]) by a specific processor to a specific address. Each event has an *issue index*, which is a sequence number relative to all events by the same processor. Furthermore, each event has a *coherence index*, which is the sequence number of the value that is read or written by the event, relative to the entire value sequence written to the targeted memory location during the execution.

3.1 Traces

Formally, let $Op = \{st, ld, il\}$, let $\mathbb{N}$ be the set of natural numbers, let $Proc = \{1, \ldots, N\}$ be a finite set of processor identifiers for some fixed bound $N \in \mathbb{N}$, let Adr be a finite set of memory addresses, and let $\mathbb{N}_0 \subseteq \mathbb{Z}$ be the set of nonnegative integers. Then we define the set of events as $Evt = Op \times Proc \times \mathbb{N} \times Adr \times \mathbb{N}_0$, and we denote elements $e \in Evt$ using the syntax $o(p, i, a, c)$, where $o \in Op$, $p \in Proc$, $i \in \mathbb{N}$ is the issue index, $a \in Adr$, and $c \in \mathbb{N}_0$ is the coherence index. We use corresponding projection functions $o(e), p(e), i(e), a(e), c(e)$ for an event e. Given a set $E \subseteq Evt$ of events, we define the following subsets for notational convenience:

(commands by proc. p)	$E(p) = \{e \in E \mid p(e) = p\}$
(load events)	$L(E) = \{e \in E \mid o(e) = ld\}$
(store events)	$S(E) = \{e \in E \mid o(e) = st\}$
(events that write)	$W(E) = \{e \in E \mid o(e) \in \{st, il\}\}$
(events that read)	$R(E) = \{e \in E \mid o(e) \in \{ld, il\}\}$
(events that write to a)	$W(E, a) = \{e \in W(E) \mid a(e) = a\}$

We call a function $f : Evt \to \mathbb{N}$ an *index* function for a subset $S' \subseteq Evt$ if $f(S') = \{1, \ldots, |S'|\}$ (including the special case where S' is empty).

DEFINITION 1 (Traces). *A* trace *is a subset $E \subseteq Evt$ satisfying*

- $(E1)$ *For all $p \in Proc$, i is an index function for $E(p)$.*
- $(E2)$ *For all $a \in Adr$, c is an index function for $W(E, a)$.*
- $(E3)$ *For all $l \in L(E)$, either $c(l) = 0$, or there exists a $w \in W(E, a(l))$ such that $c(l) = c(w)$.*

Define $\mathcal{T} \subseteq \mathcal{P}(Evt)$ to be the set of all traces. We say a trace E is a prefix *of a trace E' if $E \subseteq E'$.*

To reason about traces, we introduce binary relations $\to_p$ and $\to_c$:

- We use the program order $\to_p \subseteq Evt \times Evt$ to describe the relative order of events by the same processor. Specifically, we define $e \to_p e'$ if and only if $p(e) = p(e')$ and $i(e) < i(e')$. For any trace E, $\to_p$ is a partial order on E and a total order on $E(p)$ for all $p \in Proc$.
- We use the conflict order $\to_c \subseteq Evt \times Evt$ to describe the relative order of conflicting accesses (where we call two accesses $e, e' \in Evt$ *conflicting* if $a(e) = a(e')$ and $\{e, e'\} \cap W(Evt) \neq \emptyset$). Specifically, we define: $e \to_c e'$ if and only if $a(e) = a(e')$ and either (1) $o(e') \in W(Evt)$ and $c(e) < c(e')$, or (2) $(e, e') \in W(Evt) \times L(Evt)$ and $c(e) \leq c(e')$. The conflict

[1] We do not need to include memory fence operations because a full fence is semantically equivalent to an interlocked operation to a location that is not accessed anywhere else.

order is not actually an 'order' in the mathematical sense because it is not transitive.

3.2 Memory Models

We now proceed to define the memory models *SC* (sequential consistency) and *TSO* (total store order) using an axiomatic style. To state the definitions concisely, we define the binary relation $\rightarrow_{hb}$, called *happens-before* relation, to be the union of the program and conflict orders: $\rightarrow_{hb} = (\rightarrow_p \cup \rightarrow_c)$. Note that this definition does not make $\rightarrow_{hb}$ implicitly transitive; we will take the transitive closure $\rightarrow^*_{hb}$ explicitly if required by the context.

DEFINITION 2 (*SC*). *Define the set* $\mathcal{T}^{SC} \subseteq \mathcal{T}$ *of* sequentially consistent *traces to consist of all traces* E *that satisfy the following condition:*

(SC1) *The relation* $\rightarrow_{hb}$ *is acyclic on* E.

To define *TSO* for any given event set E, we first define the *relaxed happens-before relation* $\rightarrow_{rhb}$:

$$\rightarrow_{rhb} \;=\; \rightarrow_{hb} \;\setminus\; \{\,(e,e') \mid e \rightarrow_p e' \wedge o(e) = \mathit{st} \wedge o(e') = \mathit{ld}\}$$

Thus the $\rightarrow_{rhb}$ relation does not put a happens-before edge between a store and a subsequent load of the same processor (even if they have the same address). This reflects the existence of a store buffer: a store may globally commit after subsequent loads by the same processor, and thus not globally appear as 'happening before the load'.

DEFINITION 3 (*TSO*). *Define the set* $\mathcal{T}^{TSO} \subseteq \mathcal{T}$ *of* totally-store-ordered *traces to consist of all traces* E *that satisfy the following conditions:*

(TSO1) *The relation* $\rightarrow_{rhb}$ *is acyclic on* E.
(TSO2) *never* $(e \rightarrow_p e' \;\wedge\; e' \rightarrow_c e)$ *for any* $e, e' \in E$

The axiom (TSO2) is required to guarantee that loads correctly "snoop" the store buffer: the coherence index of a load may not be less than that of a previous store to the same address by the same processor. We establish the connection between these concise axiomatic definitions and a more verbose operational description of *SC* and *TSO* in the appendix.

3.3 Program Traces

We now formally define the set of traces $\mathcal{T}^Y_\pi$ that a program π may exhibit on a memory model $Y \in \{\mathit{SC}, \mathit{TSO}\}$. To keep our formalization light, we represent a program π abstractly by a function $next_\pi : \mathcal{T} \times \mathit{Proc} \rightarrow \mathcal{P}(\mathit{Op} \times \mathit{Adr})$. The set $next_\pi(E,p)$ describes what instructions (combinations of operation and address) may possibly be issued by processor p next, after having executed E. For a trace E, let $last(E,p)$ be the element $e \in E(p)$ such that $i(e)$ is maximal, or undefined if $E(p) = \emptyset$. We say that a program π is *locally deterministic* if for all $(E,p) \in \operatorname{dom} next_\pi$, we have (1) $|next_\pi(E,p)| \leq 1$, and (2) for all prefixes $E' \subseteq E$ such that $last(E',p) = last(E,p)$, we have $next_\pi(E,p) = next_\pi(E',p)$. In the following, we will assume without further mention that all programs are locally deterministic. For a trace $E \in \mathcal{T}$, define the set of possible successor events under program π as

$$\begin{aligned} succ_\pi(E) = \{e \in (\mathit{Evt} \setminus E) \mid (E \cup \{e\} \in \mathcal{T}) \\ \text{and } next_\pi(E, p(e)) = (o(e), a(e))\}. \end{aligned}$$

DEFINITION 4 (Program Traces). *For a program* π *and memory model* $Y \in \{\mathit{SC}, \mathit{TSO}\}$*, define the set of traces* $\mathcal{T}^Y_\pi$ *inductively as the smallest set satisfying (i)* $\emptyset \in \mathcal{T}^Y_\pi$*, and (ii) for all* $E \in \mathcal{T}^Y_\pi$ *and* $e \in succ_\pi(E)$ *such that* $E \cup \{e\} \in \mathcal{T}^Y$*, we have* $E \cup \{e\} \in \mathcal{T}^Y_\pi$.

DEFINITION 5 (Store Buffer Safety). *The program* π *is called* store buffer safe *if and only if* $\mathcal{T}^{TSO}_\pi = \mathcal{T}^{SC}_\pi$.

4. Solution

We now describe how we can check store buffer safety by exploring $\mathcal{T}^{SC}_\pi$ only. The idea is to look for *borderline traces* which are defined as follows.

DEFINITION 6 (Borderline Trace). *A sequentially consistent trace* $E \in \mathcal{T}^{SC}_\pi$ *of a program* π *is called a* borderline trace *if there exists an* $e \in succ_\pi(E)$ *such that* $E \cup \{e\} \in (\mathcal{T}^{TSO}_\pi \setminus \mathcal{T}^{SC}_\pi)$.

THEOREM 7. *A program* π *is store buffer safe if and only if it has no borderline traces.*

PROOF. If $E \in \mathcal{T}^{SC}_\pi$ is a borderline trace, then there exists a trace $E \cup \{e\} \in (\mathcal{T}^{TSO}_\pi \setminus \mathcal{T}^{SC}_\pi)$ implying $\mathcal{T}^{SC}_\pi \neq \mathcal{T}^{TSO}_\pi$. Conversely, assume $\mathcal{T}^{SC}_\pi \neq \mathcal{T}^{TSO}_\pi$. Because $\mathcal{T}^{SC}_\pi \subseteq \mathcal{T}^{TSO}_\pi$, there must exist $E \in (\mathcal{T}^{TSO}_\pi \setminus \mathcal{T}^{SC}_\pi)$. By construction of $\mathcal{T}^{TSO}_\pi$, there exist traces $E_0, \ldots, E_n \in \mathcal{T}^{TSO}_\pi$ and events $e_1, \ldots, e_n$ such that $E_0 = \emptyset$, $\{e_k\} = E_k \setminus E_{k-1}$, and $E_n = E$. Because $E_n \notin \mathcal{T}^{SC}_\pi$ but $E_0 \in \mathcal{T}^{SC}_\pi$, there exists a minimal k such that $E_k \notin \mathcal{T}^{SC}_\pi$. This implies that $E_{k-1} \in \mathcal{T}^{SC}_\pi$ and E_{k-1} is a borderline trace (because $E_{k-1} \cup \{e_k\} \in (\mathcal{T}^{TSO}_\pi \setminus \mathcal{T}^{SC}_\pi)$). □

The following cycle characterization lemma provides an efficient method to detect borderline traces. For a trace E, let $lastR(E,p)$ be the element $e \in E(p) \cap R(E)$ such that $i(e)$ is maximal, or be undefined if $(E(p) \cap R(E)) = \emptyset$; and let $write(E,a,c)$ denote the element $e \in W(E,a)$ such that $c(e) = c$ if it exists, or be undefined otherwise.

LEMMA 8 (Cycle Characterization). *Let* $E \in \mathcal{T}^{SC}_\pi$ *be a sequentially consistent trace of* π*, and let* $e = o(p,i,a,c) \in succ_\pi(E)$*. Let* $E' = E \cup \{e\}$*. Then:*

(1) $E' \notin \mathcal{T}^{SC}_\pi$ *if and only if* $o = \mathit{ld}$ *and* $write(E,a,c+1) \rightarrow^*_{hb} last(E,p)$.

(2) $E' \notin \mathcal{T}^{TSO}_\pi$ *if and only if* $o = \mathit{ld}$ *and either*
(i) $write(E,a,c+1) \rightarrow^*_{rhb} lastR(E,p)$*, or*
(ii) there exists $c' > c$ *such that* $p(write(E,a,c')) = p$.

PROOF. **(1⇐).** If $o = \mathit{ld}$ and $write(E,a,c+1) \rightarrow^*_{hb} last(E,p)$, then

$$e \rightarrow_c write(E,a,c+1) \rightarrow^*_{hb} last(E,p) \rightarrow_p e$$

which forms a $\rightarrow_{hb}$-cycle, implying $E' \notin \mathcal{T}^{SC}$ by (SC1), and thus $E' \notin \mathcal{T}^{SC}_\pi$. **(2⇐).** either (i) or (ii) must hold; if (i) holds, we proceed as in case (1⇐): we use $e \rightarrow_c write(E,a,c+1)$ and $lastR(E,p) \rightarrow_p e$ to construct a cycle (this time, a $\rightarrow_{rhb}$-cycle) which implies $E' \notin \mathcal{T}^{TSO}$ by (TSO1), and thus $E' \notin \mathcal{T}^{TSO}_\pi$. If (ii) holds, then either $write(E,a,c') \rightarrow_p e$ or $e \rightarrow_p write(E,a,c')$; but the latter is impossible because both E and E' are traces (specifically, because i is an index function on both $E(p)$ and $E'(p)$). Therefore, $write(E,a,c') \rightarrow_p e$. Along with $e \rightarrow_c write(E,a,c')$ we conclude $E' \notin \mathcal{T}^{TSO}$ by (TSO2), and thus $E' \notin \mathcal{T}^{TSO}_\pi$. **(1⇒).** Assume $E' \notin \mathcal{T}^{SC}_\pi$. Then $E' \notin \mathcal{T}^{SC}$ (by Def. 4(ii)), which means (SC1) does not hold: specifically, $E \cup \{e\}$ has a $\rightarrow_{hb}$-cycle. Because $\rightarrow_{hb}$ is acyclic on E (because $E \in \mathcal{T}^{SC}$), it must be of the form $e \rightarrow_{hb} e_1 \rightarrow_{hb} \ldots \rightarrow_{hb} e_n \rightarrow_{hb} e$ where all $e_k \in E$ and $n \geq 1$. Now, $e \rightarrow_{hb} e_1$ by definition implies that either $e \rightarrow_p e_1$ or $e \rightarrow_c e_1$. As reasoned earlier, it can not be the case that $e \rightarrow_p e_1$ (because E and E' are both traces), thus $e \rightarrow_c e_1$. This implies that $o = \mathit{ld}$ (because c is an index function on both $W(E,a)$ and $W(E',a)$). Because e is a load and $e \rightarrow_c e_1$, we know $o(e_1) \in \{\mathit{st}, \mathit{il}\}$, $a(e_1) = a$ and $c(e_1) > c$, and thus either $write(E,a,c+1) = e_1$ or $write(E,a,c+1) \rightarrow_c e_1$. Therefore $write(E,a,c+1) \rightarrow^*_{hb} e_n$. Now, it can not be the case that $e_n \rightarrow_c e$ (otherwise $e_n \rightarrow^*_c e_1$ which creates a $\rightarrow_{hb}$-cycle within E, contradicting $E \in \mathcal{T}^{SC}_\pi$), thus

```
1  function is_store_buffer_safe(e₁e₂...eₙ)
2                                         returns boolean {
3    var k,p,a,c : ℕ; var E : 𝒯;
4    E := ∅;
5    for (k := 1; k <= n; k++) {
6      if (o(eₖ) = ld) {
7        p := p(eₖ); a := a(eₖ); c := c(eₖ);
8        while (c > 0) {
9          if (p = p(write(E,a,c)))
10           break;
11         if (write(E,a,c) →*_rhb lastR(E,p))
12           break;
13         if (write(E,a,c) →*_hb last(E,p))
14           return false;
15         c := c - 1;
16       }
17     }
18     E := E ∪ eₖ;
19   }
20   return true;
21 }
```

Figure 2. Our algorithm to monitor store buffer safety in a given interleaving.

$e_n \rightarrow_p e$. Therefore, either $e_n = \mathit{last}(E,p)$ or $e_n \rightarrow_p \mathit{last}(E,p)$. We can thus conclude that $\mathit{write}(E,a,c+1) \rightarrow^*_{\mathit{hb}} \mathit{last}(E,p)$ as desired. **(2⇒).** If $E' \notin \mathcal{T}^{\mathit{TSO}}_\pi$ then $E' \notin \mathcal{T}^{\mathit{TSO}}$ (by Def. 4(ii)). Thus either (TSO1) or (TSO2) must be violated. First, assume that E' does not satisfy (TSO1). Just as in (1⇒) (but using the relation $\rightarrow_{\mathit{rhb}} \subseteq \rightarrow_{\mathit{hb}}$), we conclude that there exists a cycle of the form $e \rightarrow_{\mathit{rhb}} e_1 \rightarrow_{\mathit{rhb}} \dots \rightarrow_{\mathit{rhb}} e_n \rightarrow_{\mathit{rhb}} e$, that $e \rightarrow_c e_1$, that $o = \mathit{ld}$, that $\mathit{write}(E,a,c+1) \rightarrow^*_{\mathit{rhb}} e_n$, and that $e_n \rightarrow_p e$. The latter implies that $o(e_n) \neq \mathit{st}$ (otherwise not $e_n \rightarrow_{\mathit{rhb}} e$), and therefore either $e_n = \mathit{lastR}(E,p)$ or $e_n \rightarrow_{\mathit{rhb}} \mathit{lastR}(E,p)$. Thus condition (i) is satisfied. Next, assume that E' does not satisfy (TSO2). Because E does, and because we know that not $e \rightarrow_p e'$ for any $e' \in E$ (because E and E' are both traces), there must exist an $e' \in E$ such that $e' \rightarrow_p e$ and $e \rightarrow_c e'$. This implies $o(e) = \mathit{ld}$ (because c is an index function on both $W(E,a)$ and $W(E',a)$). Because e is a load and $e \rightarrow_c e'$, we know $o(e') \in \{\mathit{st}, \mathit{il}\}$, $a(e') = a$ and $c(e') > c$. Thus, condition (ii) is satisfied with $c' = c(e')$. □

4.1 Monitor Algorithm

Fig. 2 shows our implementation of a monitor that can monitor store buffer safety in any interleaved execution of the program. It processes the events in the sequence in order (and can thus be used online or offline) and reports any detected borderline traces. We now qualify the soundness and completeness of this monitor. For a sequence $w = e_1 \dots e_n \in \mathit{Evt}^*$ of events, let $E_w = \{e_1, \dots e_n\}$. The sequence w is called an *interleaving* of a program π if (1) the e_k are pairwise distinct, (2) $E_w \in \mathcal{T}^{\mathit{SC}}_\pi$, (3) $e_x \rightarrow_{\mathit{hb}} e_y \implies x < y$, and (4) $\mathit{next}_\pi(E_w,p) = \emptyset$ for all $p \in \mathit{Proc}$.

THEOREM 9 (Soundness). *If an an interleaving w of program π is reported unsafe by our monitor, then π is not store buffer safe.*

PROOF. Assume `is_store_buffer_safe(w)` returns false for $w = e_1 \dots e_n$. Let E, k, p, i, a and c' be the values of the program variables `E`, `k`, `p`, `i`, `a`, and `c` at the time of the return, respectively. Then $E = \{e_1, \dots, e_{k-1}\}$, and $e_k = \mathit{ld}(p,i,a,c)$ for some c. Let $e = e_k$, and let $e' = \mathit{ld}(p,i,a,c'-1)$. We now argue that $E' = E \cup \{e'\} \in (\mathcal{T}^{\mathit{TSO}}_\pi \setminus \mathcal{T}^{\mathit{SC}}_\pi)$, which implies that E is a borderline trace and thus $\mathcal{T}^{\mathit{SC}}_\pi \neq \mathcal{T}^{\mathit{TSO}}_\pi$ by Theorem 7 as desired. First, note that $e' \in \mathit{succ}_\pi(E)$ because $E \cup \{e\} \in \mathcal{T}^{\mathit{SC}}_\pi$ implies $E \cup \{e'\} \in \mathcal{T}$ and $(o,a) \in \mathit{next}_\pi(E,p)$ (using that π is locally deterministic). We can thus enlist the help of Lemma 8 to show $E' \in (\mathcal{T}^{\mathit{TSO}}_\pi \setminus \mathcal{T}^{\mathit{SC}}_\pi)$. First, because the program returned at line 14, we know $\mathit{write}(E,a,c') \rightarrow^*_{\mathit{hb}} \mathit{last}(E,p)$, which implies $E' \notin \mathcal{T}^{\mathit{SC}}_\pi$ by Lemma 8, part (1). Second, because the program did not break at line 12 right before returning on line 14, we know that not $(\mathit{write}(E,a,c') \rightarrow^*_{\mathit{rhb}} \mathit{lastR}(E,p))$. Moreover, because the while loop was not broken at line 10, we know that $p(\mathit{write}(E,a,c'')) \neq p$ for all $c'' \geq c'$. By Lemma 8, part (2) we conclude that $E' \in \mathcal{T}^{\mathit{TSO}}_\pi$. □

As for completeness, we clearly cannot detect all borderline traces by looking at a single interleaving w only. However, it is possible to detect them reliably by checking a sufficient set of interleavings. Specifically, we call a set of interleavings $I \subseteq \mathit{Evt}^*$ a *representative* for program π if for all $E \in \mathcal{T}^{\mathit{SC}}_\pi$ there exists an interleaving $w \in I$ such that $E \subseteq E_w$ and there are no $\rightarrow_{\mathit{hb}}$-edges from $E_w \setminus E$ into E.

THEOREM 10 (Completeness). *Let I be a representative set of interleavings of a program π. Then, if π is not store buffer safe, our monitor will detect it on some interleaving $w \in I$.*

PROOF. By Theorem 7, we know that $\mathcal{T}^{\mathit{SC}}_\pi \neq \mathcal{T}^{\mathit{TSO}}_\pi$ implies that there exists a borderline trace $E \in \mathcal{T}^{\mathit{SC}}_\pi$. Thus there exists an element $e = o(p,i,a,c) \in \mathit{Evt}$ such that $E' = (E \cup \{e\}) \in \mathcal{T}^{\mathit{TSO}}_\pi \setminus \mathcal{T}^{\mathit{SC}}_\pi$. Because I is representative, it must contain an interleaving $w = e_1 \dots e_n$ such that $E \subseteq E_w$ is a prefix. Because $(o(e), a(e)) \in \mathit{next}_\pi(E,p)$, there must be a k such that $p(e_k) = p$ and $i(e_k) = i$ (otherwise $\mathit{last}(E_w,p) = \mathit{last}(E,p)$ and thus $\mathit{next}_\pi(E,p) = \mathit{next}_\pi(E_w,p)$, contradicting $\mathit{next}_\pi(E_w,p) = \emptyset$). We now claim that if the algorithm reaches the k-th iteration, it must return false (if it returns prior to that, it also returns false and we are satisfied). Let $E_k = \{e_1, \dots, e_{k-1}\}$. By Lemma 8, part (1), we know that $\mathit{write}(E,a,c+1) \rightarrow^*_{\mathit{hb}} \mathit{last}(E,p)$ within E. Now, by the choice of k, we know $E(p) = E_k(p)$, thus $\mathit{last}(E,p) = \mathit{last}(E_k,p)$, and because w is an interleaving (respects $\rightarrow_{\mathit{hb}}$), this implies $\mathit{write}(E_k,a,c+1) \rightarrow^*_{\mathit{hb}} \mathit{last}(E_k,p)$ within E_k. Moreover, we know that $c(e_k) \geq (c+1)$ because w is an interleaving and $\mathit{write}(E_k,a,c+1)$ appears before e_k in w. Thus, the while loop (which assigns $c(e_k)$ to the variable `c` initially, and then keeps decrementing it) must eventually return true at line 14 unless it is broken at either line 10 or line 12. But that is not possible, for the following reasons. First, suppose line 10 breaks. Let c' be the value of the variable `c` at that time; then $c+1 \leq c' \leq c(e_k)$ and $p(\mathit{write}(E_k,a,c')) = p$. Now, because $E(p) = E_k(p)$, we know $\mathit{write}(E_k,a,c') \in E$. Thus, $\mathit{write}(E,a,c') = \mathit{write}(E_k,a,c')$, implying $p(\mathit{write}(E,a,c')) = p$ which in turn implies $E' \notin \mathcal{T}^{\mathit{TSO}}_\pi$ by Lemma 8, part (2ii), contradicting the assumption. Next, suppose line 12 breaks. Let c' be the value of the variable `c` at that time; then $c+1 \leq c' \leq c(e_k)$ and $\mathit{write}(E_k,a,c') \rightarrow^*_{\mathit{rhb}} \mathit{lastR}(E_k,p)$ within E_k. Now, because $E(p) = E_k(p)$, $\mathit{lastR}(E_k,p) = \mathit{lastR}(E,p)$. Because there are no $\rightarrow_{\mathit{hb}}$-edges (and thus no $\rightarrow_{\mathit{rhb}}$-edges) from E_w into E, this implies that $\mathit{write}(E,a,c') \rightarrow^*_{\mathit{rhb}} \mathit{lastR}(E,p)$. Because $c+1 \leq c'$, this implies $\mathit{write}(E,a,c) \rightarrow^*_{\mathit{rhb}} \mathit{lastR}(E_k,p)$, which in turn implies $E' \notin \mathcal{T}^{\mathit{TSO}}_\pi$ by Lemma 8, part (2i), contradicting the assumption. □

A stateless model checker (such as Verisoft [11] or CHESS[22]) can provide us with a representative set of interleavings if the program is bounded (we call a program *bounded* if there exists a number $M \in \mathbb{N}$ such that $|E| < M$ for all $E \in \mathcal{T}^{\mathit{SC}}_\pi$). The following theorem clarifies that this is true even if partial order reduction is employed. We call a set of interleavings $I \subseteq \mathit{Evt}^*$ a *partial-order-complete set* for program π if for all interleavings w of π, there exists a w' in I such that $E_w = E_{w'}$.

THEOREM 11. *If I is a partial-order-complete set of interleavings for a bounded program π, then it is representative for π.*

```
type       timestamp: array[2*N] of $\mathbb{N}_0$;
var        lc: array[Proc] of timestamp;
           sc: array[Proc] of timestamp;
           mc1: array[Proc][Adr] of timestamp;
           mc2: array[Adr] of timestamp;
initially  lc[*][*] = sc[*][*] = mc1[*][*][*] = mc2[*][*] = 0;
function merge(ts$_1$, ... ts$_n$ : timestamp) returns timestamp {
  return (max$_i$(ts$_i$[1]), ... , max$_i$(ts$_i$[N*2]));
}
function process_event(e : Evt) returns timestamp {
  match e with
    ld(p,i,a,c) ->
      ts := merge(lc[p], mc1[p][a]);
      ts[2*p] := ts[2*p] + 1;     // advance load count for p
      lc[p] := merge(lc[p], ts);
      mc2[a] := merge(mc2[a], ts);
    st(p,i,a,c) ->
      ts := merge(sc[p], lc[p], mc2[a]);
      ts[2*p+1] := ts[2*p+1] + 1; // advance store count for p
      forall q $\neq$ p do
        mc1[q][a] := merge(mc1[q][a], ts);
      mc2[a] := merge(mc2[a], ts);
      sc[p] := merge(sc[p], ts);
    il(p,i,a,c) ->
      ts := merge(sc[p], lc[p], mc2[a]);
      ts[2*p] := ts[2*p] + 1;     // advance load count for p
      ts[2*p+1] := ts[2*p+1] + 1; // advance store count for p
      forall q $\in$ Proc do
        mc1[q][a] := merge(mc1[q][a], ts);
      mc2[a] := merge(mc2[a], ts);
      lc[p] := merge(lc[p], ts);
      sc[p] := merge(sc[p], ts);
  return ts;
}
```

Figure 3. A vector clock for tracking the transitive closure $\rightarrow^*_{rhb}$.

PROOF. We now prove Theorem 11. The significance of this theorem arises from the fact that most stateless model checkers, such as Verisoft [11] and CHESS [22], use partial-order reduction techniques [10, 8] to only explore a subset of all interleavings of the program. Theorem 11 shows that to prove the store buffer safety of a program it is sufficient to run the monitor algorithm of Section 4.1 only on the set of executions explored by any partial-order reduction algorithm.

We will assume the most commonly used definition of dependent events in a program. Two events are *dependent* if and only if they are performed by the same thread or access the same memory location with at least one of the accesses being a write. Effectively, two events in an execution are dependent if and only if they are ordered by $\rightarrow^*_{hb}$. Two events are *independent* if they are not dependent on each other. Two interleavings w and w' are equivalent if one can be obtained from the other by iteratively commuting two consecutive independent events. Thus, w and w' are equivalent if and only if $E_w = E_{w'}$.

By definition, a partial-order-complete set for program π contains at least one representative from each equivalence class of all interleavings of π. Therefore, the set of interleavings explored by any partial-order reduction algorithm is a partial-order-complete set. Now we can prove Theorem 11.

Let E be an execution in $\mathcal{T}^{SC}_\pi$ and let I be any partial-order-complete set for π. Using Definition 4 and the fact that π is bounded, we can inductively generate a sequence of executions $E = E_1, E_2, \ldots, E_n$ such that for all $1 \leq i < n$, (1) $E_{i+1} \in \mathcal{T}^{SC}_\pi$, (2) $\exists e_i \in succ_\pi(E_i) : E_{i+1} = E_i \cup \{e_i\} \wedge \forall e \in E_i : e \rightarrow^*_{hb} e_i$, and (3) $succ_\pi(E_n) = \emptyset$. Informally, this sequence represents the execution of π from the program state at E till completion. Let w be an interleaving of π such that $E_w = E_n$. By definition, there exists an interleaving $w' \in I$ such that $E_{w'} = E_w = E_n$. Moreover, by construction, $E \subseteq E_n$ and there are no $\rightarrow_{hb}$-edges from $E_n \setminus E$ into E. Since we started with an arbitrary $E \in \mathcal{T}^{SC}_\pi$, we conclude that I is a representative set. □

4.2 Vector Clocks

The pseudocode in Fig. 2 does not detail how to decide the conditions on lines 11 and 13. While it is well known how to use vector clocks to compute the transitive closure $\rightarrow^*_{hb}$ for a given interleaving of length n in time $O(nN)$, it is not immediately clear how to do the same for $\rightarrow^*_{rhb}$. We solved this problem by generalizing vector clocks (Def. 13 below) and by engineering a vector clock instance (Fig. 3) that can compute the transitive closure $\rightarrow^*_{rhb}$ in time $O(nN^2)$.

THEOREM 12. *Let $w = e_0 \ldots e_n$ be an interleaving of some program π, and let $t_1, \ldots, t_n$ be the timestamps returned by the corresponding sequence of calls to* `process_event` *(Fig. 3). Then $e_i \rightarrow^*_{rhb} e_j$ if and only if $i \leq j$ and $t_i[k] \leq t_j[k]$ for all $k \in \{1, \ldots, 2N\}$.*

We first describe informally how this vector clock works. Our vector clock uses timestamps of a fixed width (here $2N$, where N is the maximal number of processors) and maintains a number of clocks (defined as global variables in Fig. 3). The computation of each timestamp follows the following pattern: (1) some of the clocks are read and merged, (2) some positions of the resulting vector are incremented to form the timestamp, and (3) the timestamp is merged back into some of the clocks. The following definition clarifies the conditions that underly this general mechanism ($in(e)$ and $out(e)$ represent the clock sets in step (1) and (3), respectively, and $gps(e)$ represents the set of positions in step (2)).

DEFINITION 13 (General Vector Clock). *Let Σ be a set of events, and let $\rightarrow$ be a binary relation on Σ. A* general vector clock *for $(\Sigma, \rightarrow)$ is a tuple (C, G, in, out, gps) where C is a set of clocks,*

name and description	lines of code	context bound	no. of interleavings total	borderline	falsification time [s]	verification time [s] SoBeR	CHESS
Fig. 1(b)	42	∞	10	4	< 0.1	< 0.2	< 0.2
dekker	82	1	5	4	< 0.1	< 0.2	< 0.2
(2 threads,		2	36	23	< 0.1	0.39	0.37
2 critical		3	183	50	< 0.1	1.9	1.8
sections)		4	1,219	124	< 0.1	13.2	13.0
		5	8,472	349	< 0.1	106.0	100.6
bakery	122	0	1	1	< 0.1	< 0.2	< 0.2
(2 threads,		1	25	20	< 0.1	0.47	0.43
3 critical		2	742	533	< 0.1	10.3	9.8
sections)		3	12,436	8,599	< 0.1	189.0	181.0
takequeue	374	0	3	0	not found	< 0.3	< 0.3
(2 threads,		1	47	14	0.34	0.72	0.69
6 operations)		2	402	189	0.43	5.2	4.9
		3	2,318	1,197	0.74	28.9	27.8
		4	9,147	5,321	0.84	125.5	118.9
		5	29,821	17,922	0.86	481.5	461.6

Figure 4. Experiments on a 2.2GHz Intel Core Duo laptop running Windows Vista.

G is a set of groups, in, out are functions $\Sigma \to \mathcal{P}(C)$, and gps is a function $\Sigma \to \mathcal{P}(G)$ such that the following conditions are satisfied:

(VC1) for all $\sigma \in \Sigma$, $gps(\sigma) \neq \emptyset$.

(VC2) for all $g \in G$, $\to$ is a total order on $\{\sigma \in \Sigma \mid g \in gps(\sigma)\}$.

(VC3) for all $\sigma, \sigma' \in \Sigma$, we have $(out(\sigma) \cap in(\sigma') \neq \emptyset) \Leftrightarrow (\sigma \to \sigma')$.

To conclude the proof of Theorem 12, we first connect the general definition of vector clock to Fig. 3. To do so, we define $\Sigma = \mathit{Evt}$ and $\to = \to_{rhb}$. Furthermore, we use clocks $C = \{lc[p], sc[p], mc1[p][a], mc2[a] \mid a \in Adr, p \in Proc\}$, and groups $G = \{r_1, w_1, r_2, w_2, \ldots, r_N, w_N\}$, and we define gps in such a way that $(r_p \in gps(e) \Leftrightarrow e \in R(Evt(p)))$ and $(w_p \in gps(e) \Leftrightarrow e \in W(Evt(p)))$. We then define $in(e)$ to contain the clocks that get merged into the timestamp (in the respective match clause for e in Fig. 3), and out to define the clocks to which a timestamp propagates. Now, (VC1) is satisfied because each access is a read or a write (possibly both), (VC2) is satisfied ($\to_{rhb}$ is a total order within each $R(Evt(p))$ and $W(Evt(p))$), and (VC3) is satisfied (which we can check by manual, pairwise comparison of the match cases).

We now formalize the computation performed by a general vector clock and shows that it gives correct results. Given a general vector clock as in Def. 13, define the set of *timestamps* $T = (\mathbb{N}_0)^G$, define a partial order on timestamps $t_1 \leq t_2 \Leftrightarrow (\forall g \in G : t_1(g) \leq t_2(g))$, and define the operations $merge : \mathcal{P}(T) \to T$ and $increment : T \times E \to T$ as follows: $merge(S)(g) = \max_{t \in S} t(g)$ (with corner case $merge(\emptyset)(g) = 0$), and

$$increment(t, e)(g) = \begin{cases} t(g) + 1 & \text{if } g \in gps(e) \\ t(g) & \text{otherwise.} \end{cases}$$

Given a sequence of events $\sigma_1, \ldots, \sigma_n$, we say the vector clock (C, G, in, out, gps) *computes the timestamps* $t_1, \ldots t_n \in T$ if for each clock $c \in C$ there exist timestamps $c_1, \ldots c_n \in T$ such that the following conditions are satisfied:

(t1) $c_1(g) = 0$ for all $c \in C$, $g \in G$.

(t2) $t_i = increment(merge\{c_i \mid c \in in(\sigma_i)\})$ for $1 \leq i \leq n$.

(t3) $c_{i+1} = \begin{cases} merge\{c_i, t_i\} & \text{if } c \in out(\sigma_i) \\ c_i & \text{otherwise} \end{cases}$
for $1 \leq i \leq (n-1)$.

THEOREM 14. *Let Σ be a set of events, let $\to$ be a binary relation on Σ, let $\sigma_1, \ldots, \sigma_n \in \Sigma$ be a sequence of events satisfying $(\sigma_i \to \sigma_j \Rightarrow i < j)$, and let $t_1, \ldots, t_n \in T*$ be computed by a vector clock for $(\Sigma, \to)$. Then $\sigma_i \to^* \sigma_j$ if and only if $t_i \leq t_j$.*

PROOF. In the following, let $P = \{1, \ldots, n\}$ be the set of positions. For $i \in P$, define the set $M_i = \{j \in P \mid \sigma_j \to \sigma_i\}$. To prepare for the proof, we first prove the following two properties for arbitrary $i, j \in P$ and $g \in G$:

$$t_i = increment(merge\{t_k \mid k \in M_i\}, \sigma_i) \qquad (1)$$

$$(t_i(g) \leq t_j(g) \wedge g \in gps(\sigma_i)) \Rightarrow \sigma_i \to^* \sigma_j \qquad (2)$$

To prove (1), note that $merge$ is a "flat" function, meaning that $merge(merge(A), B) = merge(A \cup B)$. Thus $merge\{c_i \mid c \in in(\sigma_i)\} = merge(\bigcup_{c \in in(\sigma_i)} merge(\{t_j \mid (j < i) \wedge c \in out(\sigma_j)\} \cup \{c_1\}) = merge\{t_j \mid (j < i) \wedge out(\sigma_j) \cap in(\sigma_i) \neq \emptyset\} = merge\{t_k \mid k \in M_i\}$. By applying this to (t2), we get (1). To prove (2), assume it is not true. Then we can pick $j \in P$ minimal such that there exist i, g such that $t_i(g) \leq t_j(g)$ and $g \in gps(\sigma_i)$, yet not $\sigma_i \to^* \sigma_j$. Clearly, this implies $i \neq j$. Now, first consider the case $g \in gps(\sigma_j)$. Because $i \neq j$ and by (VC2) and $(\sigma_i \to \sigma_j \Rightarrow i < j)$, this implies $\sigma_i \to \sigma_j$ which is a contradiction. Thus $g \notin gps(\sigma_j)$. Moreover, $t_j(g) \neq 0$ (because $t_i(g) \neq 0$ and $t_i(g) \leq t_j(g)$ by assumption). Thus, the merge in (1) can not be empty, meaning that there exists a $k \in M_j$ such that $t_j(g) = t_k(g)$. By minimality of j, $\sigma_i \to^* \sigma_k$. But this implies a contradiction.

We now prove the two directions of the theorem. ($\Rightarrow$). Assume there exist $k_1, \ldots, k_l \in P$ such that $\sigma_i = \sigma_{k_1} \to \cdots \to \sigma_{k_l} = \sigma_j$. Then $k_m \in M_{k_{m+1}}$ for $1 \leq m < l$. By (1) this implies $t_{k_1} \leq \cdots \leq t_{k_l}$ and thus $t_i \leq t_j$. ($\Leftarrow$). Assume $t_i \leq t_j$. Pick an arbitrary $c \in gps(\sigma_i)$ (which is nonempty by (VC1)). Then $t_i(g) = t_j(g)$ and thus $\sigma_i \to^* \sigma_j$ by (2). □

5. Experiments

We present experimental results for four C# programs (Fig. 4). The largest one (takequeue) implements a low-lock datastructure and is part of a concurrency library at Microsoft. For all programs, Sober (1) *falsified* the original version (found that it is not store buffer safe), and (2) *verified* a fixed version (which we obtained by adding more memory fences whenever Sober showed us a borderline trace) up to some bound on the number of preemptions [22] (column 2).

We make two observations. First, a large percentage of interleavings trip the monitor (columns 3,4). Therefore, a violation is found quickly (column 5). This indicates that our monitor may be useful for falsification even in a plain testing setup (without doing exhaustive space exploration). Second, when verifying a correct program, the number of interleavings and the verification time increase dramatically with the context bound as usual [22]; however, the overhead by the store buffer safety monitor is fairly low in practice (columns 6,7), indicating that it makes sense to turn it on by default within the CHESS tool.

6. Conclusions and Future Work

We have presented a novel method to verify store buffer safety using a non-intrusive monitor that is run alongside sequentially consistent executions of the program. We have demonstrated that this method is scalable, automatic and precise enough to find store-buffer-related bugs in realistic low-lock code, such as concurrency libraries.

As future work, we consider including memory model relaxations other than store buffers, and we plan to apply our monitor to larger execution traces.

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[2] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[3] S. Burckhardt, R. Alur, and M. Martin. Bounded verification of concurrent data types on relaxed memory models: A case study. In *Computer-Aided Verification (CAV)*, LNCS 4144, pages 489–502. Springer, 2006.

[4] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[5] Compaq Computer Corporation. *Alpha Architecture Reference Manual*, 4th edition, January 2002.

[6] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[7] X. Fang, J. Lee, and S. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *International Conference on Supercomputing (ICS)*, pages 285–294, 2003.

[8] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL 05: Principles of Programming Languages*. ACM Press, 2005.

[9] B. Frey. *PowerPC Architecture Book v2.02*. International Business Machines Corporation, 2005.

[10] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer-Verlag, 1996.

[11] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.

[12] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Computer-Aided Verification (CAV)*, LNCS 3114, pages 401–413, 2004.

[13] M. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31(8):28–34, 1998.

[14] T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Formal Methods (FM)*, LNCS 4085, pages 476–491. Springer, 2006.

[15] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, November 2006.

[16] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*, August 2007.

[17] International Business Machines Corporation. *z/Architecture Principles of Operation*, first edition, December 2000.

[18] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

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

[20] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, pages 378–391, 2005.

[21] V. Morrison. Understand the impact of low-lock techniques in multithreaded apps. *MSDN Magazine*, 20(10), October 2005.

[22] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Programming Language Design and Implementation (PLDI)*, pages 446–455, 2007.

[23] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 34–41, 1995.

[24] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[25] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

[26] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. In *International Conference on Formal Engineering Methods (ICFEM)*, LNCS 3308, pages 30–45. Springer, 2004.

[27] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Rigorous concurrency analysis of multithreaded programs. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, 2004.

[28] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

A. Operational Memory Models

In the following, we present our operational models for *SC* and *TSO*. Operational memory models describe what traces are valid by specifying an automaton which emits events as it transitions. Traces are valid with respect to the operational model if and only if there exists an accepting run of the automaton such that the emitted events match the events in the trace. We then prove (Thm. 16 and 17) that the axiomatic definitions of *SC* and *TSO* (Def. 2 and 3) are equivalent to the operational models.

We now describe this concept more formally. We call our labeled transition systems *trace automata* (note that they are not required to be finite), and work over a fixed set *Evt* of events (we defined this set previously). A trace automaton over *Evt* is a tuple $A = (S, T, \mathit{initial}, \mathit{accept}, \mathit{guard}, \mathit{apply})$ where S is a set of states, $T \supseteq \mathit{Evt}$ is a set of transitions (transitions in $T \setminus \mathit{Evt}$ are called *internal*), $\mathit{initial} : S \rightarrow \mathit{bool}$ and $\mathit{accept} : S \rightarrow \mathit{bool}$ are predicates that characterize the initial and accepting states, *guard* maps each transition $t \in T$ to a boolean function $\mathit{guard}\langle t \rangle : S \rightarrow \mathit{bool}$ (describing the guard of t), and *apply* maps each transition $t \in T$ to a function $\mathit{apply}\langle t \rangle : S \rightarrow S$ (describing the effect of t on the state). We then say a sequence $w = w_0 \dots w_n \in T^*$ is *an accepted run* by A if there exist states $s_0, \dots, s_{n+1} \in S$ such that all of the following hold:

- $\mathit{initial}(s_0) = \mathit{true}$
- $\mathit{accept}(s_{n+1}) = \mathit{true}$
- $\mathit{guard}\langle w_i \rangle (s_i) = \mathit{true}$ for all $0 \leq i \leq n$
- $\mathit{apply}\langle w_i \rangle (s_i) = s_{i+1}$ for all $0 \leq i \leq n$

We define $\mathcal{L}(A) \subseteq T^*$ to be the set of all accepted runs of A. For a sequence $w \in T^*$, let $(w \uparrow \mathit{Evt})$ be the subsequence of w that contains only the elements that are in *Evt*. We then define the set of memory traces of an event automaton A as $\mathcal{T}(A) = \{w \uparrow \mathit{Evt} \mid w \in \mathcal{L}(A)\}$.

We now define the trace automata A_{SC} and A_{TSO}. The automaton A_{SC} has no internal transitions; the automaton A_{TSO} has internal transitions $st^c(p, i, a, c)$ that correspond to the time at which the store $st(p, i, a, c)$ commits globally (which may happen at a later time than the issuing of the store).

DEFINITION 15. *Define the automaton A_{SC} as shown in Fig. 5, and define the automaton A_{TSO} as shown in Fig. 6.*

We now prove that the axiomatic definitions of *SC* and *TSO* (Def. 2 and 3) are equivalent to the operational models.

THEOREM 16. $\mathcal{T}(A_{SC}) = \mathcal{T}^{SC}$.

PROOF.
We prove equality by showing mutual containment, starting with $\mathcal{T}(A_{SC}) \subseteq \mathcal{T}^{SC}$. Let $E \in \mathcal{T}(A_{SC})$. By definition, this implies that there exists an accepting run $w = w_0 \dots w_n \in \mathit{Evt}^*$ of A_{SC}, with corresponding state sequence $s_0, \dots, s_{n+1} \in S$, such that $E = \{w_0, \dots, w_n\}$. We now need to show that E is indeed a trace (Def. 1) and that it is sequentially consistent (Def. 2).

First, we show that E is indeed a trace, that is, satisfies conditions (E1), (E2) and (E3) of Def. 1. To do so, we first look at how the `i[p]` component of the state is updated during the run. A quick look at the pseudocode in Fig. 5 reveals that `i[p]` starts out as 0 and gets incremented by each w_k for which $p(w_k) = p$, and for each such w_k we have $i(w_k) = (s_k.\texttt{i[p]}) + 1$. This implies that for all p, i is an index function on $\{w_k \mid p(w_k) = p\}$ and thus on $E(p)$ (note that the events w_k are pairwise unequal because they either go to different processors, or receive a different processor index). (E1) is thus satisfied. Similarly, we can look at the events that modify the `m[a]` component of the state. We see that `m[a]` starts out as 0 and gets incremented by each w_k such that $a(w_k) = a$ and $o(w_k) \in \{\mathit{st}, \mathit{il}\}$, and for each such w_k we have $c(w_k) = (s_k.\texttt{m[a]}+1)$. This implies that for all a, c is an index function on $\{w_k \mid (a(w_k) = a) \wedge (o(w_k) \in \{\mathit{st}, \mathit{il}\})\}$ and thus on $W(E, a)$. (E2) is thus satisfied. Finally, we can see that for each w_k satisfying $o(w_k) = \mathit{ld}$, the coherence index $c(w_k)$ matches the current `m[a]`, and is thus either equal to 0 (the initial value of `m[a]`) or to the value last written to `m[a]` by a previous store or interlocked event (and thus equal to the coherence index of that event). (E3) is thus satisfied and E is indeed a trace.

Next, we need to show that E is sequentially consistent according to Definition 2, that is, we need to show that $\rightarrow_{hb}$ is acyclic on E. Because `i[p]` and `m[a]` grow monotonically, we know that for any w_i, w_j, it must be the case that $w_i \rightarrow_p w_j \Rightarrow i < j$ and $w_i \rightarrow_c w_j \Rightarrow i < j$. Therefore, $w_i \rightarrow_{hb} w_j \Rightarrow i < j$. Thus, $\rightarrow_{hb}$ is acyclic on $\{w_1, \dots, w_n\} = E$. This concludes the proof of $\mathcal{T}(A_{SC}) \subseteq \mathcal{T}^{SC}$.

We now proceed to the second half of the proof and show $\mathcal{T}^{SC} \subseteq \mathcal{T}(A_{SC})$. Given any $E \in \mathcal{T}^{SC}$, let $w = w_0 \dots w_n \in \mathit{Evt}^*$ be a sequence such that $E = \{w_0, \dots, w_n\}$ and such that $i < j$ whenever $w_i \rightarrow_{hb} w_j$ (such a sequence always exists because E is finite and $\rightarrow_{hb}$ is acyclic by (SC1)). We then claim that w is an accepted run by A_{SC}. To show that this is indeed true, consider the state sequence $s_0, s_1, \dots, s_n$ such that $\mathit{initial}_{SC}(s_0) = \mathit{true}$ and $\mathit{apply}_{SC}\langle w_i \rangle (s_i) = s_{i+1}$. Then it remains to show that for all i, $\mathit{guard}\langle w_i \rangle (s_i) = \mathit{true}$. We do this by examining the conditions that appear in these guards individually.

- The condition `i[p]==i-1` (which appears in all guards of events in $E(p)$) is guaranteed to be satisfied because `i[p]` starts as zero, is incremented by each $w_i \in E(p)$, is not modified by any events not in $E(p)$, and (E1) guarantees that i is an index function on $E(p)$.
- The condition `m[a]==c-1` (which appears in the guards of events in $W(E, a)$) is guaranteed to be satisfied because `m[a]` starts as zero, is incremented by each $w_l \in W(E, a)$, is not modified by any events not in $W(E, a)$,and (E2) guarantees that c is an index function on $W(E, a)$.
- The condition `m[a]==c` (which appears in the guards of events in $L(E, a)$) is guaranteed for the following reason. Let $w_x = \mathit{ld}(p, i, a, c) \in E$ the load event in question. Note that `m[a]` is only modified by events in $W(E, a)$. Now, by (E3), either $c = 0$, or there exists an event $\mathit{st}(p', i', a, c) \in E$. If $c = 0$, then for all $w_y \in W(E, a)$, we have $w_x \rightarrow_c w_y$, which implies $x < y$. Thus `m[a]` still has the initial value 0 in state s_x, and the condition is satisfied. On the other hand, if there is an event $w_y = \mathit{st}(p', i', a, c) \in E$, then by the same argument, $y < x$. Moreover, by the same argument again, all other writes $w_z \in W(E, a)$ must satisfy either $z < y$ or $x < z$, which implies that the condition `m[a]==c` is satisfied in state s_i.

□

THEOREM 17. $\mathcal{T}(A_{TSO}) = \mathcal{T}^{TSO}$.

PROOF.
First, let us define the functions $\mathit{issue}, \mathit{commit} : \mathit{Evt} \rightarrow T_{TSO}$ as follows:

$$\begin{array}{rcl} \mathit{issue}(o(p,i,a,c)) & = & o(p,i,a,c) \\ \mathit{commit}(o(p,i,a,c)) & = & \left\{ \begin{array}{ll} o(p,i,a,c) & \text{if } o \neq \mathit{st} \\ st^c(p,i,a,c) & \text{if } o = \mathit{st} \end{array} \right. \end{array}$$

These functions help us reason about issue and commit transitions corresponding to events.

A_{SC} = $(S_{SC}, T_{SC}, \mathit{initial}_{SC}, \mathit{accept}_{SC}, \mathit{guard}_{SC}, \mathit{apply}_{SC})$

T_{SC} = $\{\mathit{st}, \mathit{ld}, \mathit{il}\} \times \mathit{Proc} \times \mathbb{N} \times \mathit{Adr} \times \mathbb{N}_0$

S_{SC} =
```
{
  m : array[Adr] of ℕ;
  i : array[Proc] of ℕ;
}
```

$\mathit{initial}_{SC}$ =
```
(forall a : m[a] == 0)
and (forall p : i[p] == 0)
```

accept_{SC} = `true`

t	$\mathit{guard}_{SC}\langle t\rangle$	$\mathit{apply}_{SC}\langle t\rangle$
st(p,i,a,c)	`(i[p] == i-1)` `and (m[a] == c-1)`	`i[p] := i;` `m[a] := c;`
ld(p,i,a,c)	`(i[p] == i-1)` `and (m[a] == c)`	`i[p] := i;`
il(p,i,a,c)	`(i[p] == i-1)` `and (m[a] == c-1)`	`i[p] := i;` `m[a] := c;`

Figure 5. The trace automaton A_{SC}.

A_{TSO} = $(S_{TSO}, T_{TSO}, \mathit{initial}_{TSO}, \mathit{accept}_{TSO}, \mathit{guard}_{TSO}, \mathit{apply}_{TSO})$

T_{TSO} = $\{\mathit{st}, \mathit{st}^c, \mathit{ld}, \mathit{il}\} \times \mathit{Proc} \times \mathbb{N} \times \mathit{Adr} \times \mathbb{N}_0$

S_{TSO} =
```
{
  m : array[Adr] of ℕ;
  ml : array[Proc,Adr] of ℕ;
  i : array[Proc] of ℕ;
  B : array[Proc] of FIFOQueue of Evt;
}
```

$\mathit{initial}_{TSO}$ =
```
(forall a : m[a] == 0)
and (forall p,a : ml[p,a] == 0)
and (forall p : i[p] == 0)
and (forall p : B[p].length() == 0)
```

accept_{TSO} = `(forall p : B[p].length() == 0)`

t	$\mathit{guard}_{TSO}\langle t\rangle$	$\mathit{apply}_{TSO}\langle t\rangle$
st(p,i,a,c)	`i[p] == i-1`	`i[p] := i;` `ml[p,a] := c;` `B[p].add(st(p,i,a,c));`
st^c(p,i,a,c)	`(B[p].peek()` `== st(p,i,a,c))` `and (m[a] == c-1)`	`m[a] := c;` `B[p].pop();`
ld(p,i,a,c)	`(i[p] == i-1) and` `c == max { m[a], ml[p,a] }`	`i[p] := i;`
il(p,i,a,c)	`(i[p] == i-1) and` `(m[a] == c-1) and` `B[p].length() == 0`	`i[p] := i;` `m[a] := c;`

Figure 6. The trace automaton A_{TSO}.

We prove equality by showing mutual containment, starting with $\mathcal{T}(A_{TSO}) \subseteq \mathcal{T}^{TSO}$. Let $E \in \mathcal{T}(A_{TSO})$. By definition, this implies that there exists an accepting run $w = w_0 \ldots w_n \in T^*_{TSO}$ of A_{TSO}, with corresponding state sequence $s_0, \ldots, s_{n+1} \in S$, such that $E = \{w_0, \ldots, w_n\} \cap \mathit{Evt}$. We now need to show that E is indeed a trace (Def. 1) and that it is totally-store-ordered (Def. 2).

First, we show that E is indeed a trace, that is, satisfies conditions (E1), (E2) and (E3) of Def. 1. To do so, we first look at how the `i[p]` component of the state is updated during the run. Just as in the proof of Thm. 16, we conclude that (E1) is satisfied (note that the internal transitions $\mathit{st}^c(p, i, a, c)$ do neither read nor update `i[p]`). Next, we observe that the $\mathit{st}(p, i, a, c)$ and $\mathit{st}^c(p, i, a, c)$ events occur in matching pairs, with *st* preceding st^c. Too see why, consider the variable `B[p]` for each p. It starts out empty and ends empty (because of the accepting condition). Because `B[p]` is a FIFO queue, we can match the `add` and `pop` calls, which gives the claimed match. Now, just as in the proof of Thm. 16, we figure that c is an index function on $\{w_k \mid (a(w_k) = a) \wedge (o(w_k) \in \{\mathit{st}^c, \mathit{il}\})\}$, and because of the matching this implies that c is an index function on $\{w_k \mid (a(w_k) = a) \wedge (o(w_k) \in \{\mathit{st}, \mathit{il}\})\}$ which implies (E2). Finally, we can see that for each w_k satisfying $o(w_k) = \mathit{ld}$, the coherence index $c(w_k)$ matches either the current `m[a]` or the current `ml[a]`. It is thus either equal to 0 (the initial value of `m[a]` and `ml[a]`.) or to the value last written to `m[a]` or `ml[a]` by a previous store or interlocked event (and thus equal to the coherence index of that event). (E3) is thus satisfied and E is indeed a trace.

Next, we need to show that E satisfies conditions (TSO1) and (TSO2). Let $W = \{w_0, \ldots, w_n\}$, and define the total order $<$ on W by $w_i < w_j \Leftrightarrow i < j$. Because (as remarked earlier) $\mathit{st}(p, i, a, c)$ and $\mathit{st}^c(p, i, a, c)$ events occur in matching pairs, we know $\mathit{commit}(E) \subseteq W$ and for all $e \in E$, $\mathit{commit}(e) \geq e$. We now prove that $\rightarrow_{\mathit{rhb}}$ is acyclic on E by showing the following claim:

$$\text{for all } e, e' \in E : (e \rightarrow_{\mathit{rhb}} e') \Rightarrow (\mathit{commit}(e) < \mathit{commit}(e'))$$

We prove it by doing a case distinction on $e \rightarrow_{\mathit{rhb}} e'$.

- $[e \to_p e']$. Then $e, e' \in E(p)$ for some p. Because `i[p]` grows monotonically and is incremented by the events in $E(p)$, we know $e < e'$. Now, do another case distinction.
  - $[o(e) \neq \mathit{st}]$ In this case $\mathit{commit}(e) = e < e' \leq \mathit{commit}(e')$.
  - $[o(e) = o(e') = \mathit{st}]$ e and e' are pushed into the FIFO buffer `B[p]` and popped by $\mathit{commit}(e)$ and $\mathit{commit}(e')$, so their relative order is preserved: $e < e' \Rightarrow \mathit{commit}(e) < \mathit{commit}(e')$.
  - $[o(e) = \mathit{st}$ and $o(e') = \mathit{ld}]$ This case is not possible by definition of $\to_{\mathit{rhb}}$.
  - $[o(e) = \mathit{st}$ and $o(e') = \mathit{il}]$ The guard condition of e' requires that the buffer `B[p]` be empty, therefore $e < \mathit{commit}(e) < e' = \mathit{commit}(e')$.
- $[e \to_c e']$. Then $a(e) = a(e')$ for some $a \in \mathit{Adr}$. We do a further case distinction.
  - $[e, e' \in W(E, a)]$ Because `m[a]` grows monotonically and is incremented by the events in $\mathit{commit}(W(E, a))$, we can see that $e \to_c e' \Rightarrow \mathit{commit}(e) < \mathit{commit}(e')$.
  - $[o(e) = \mathit{ld}$ and $e' \in W(E, a)]$ Because $c(e) < c(e')$, and because `m[a]` grows monotonically, it can not be the case that $\mathit{commit}(e')$ happens before e. Thus $\mathit{commit}(e) = e < \mathit{commit}(e')$.
  - $[o(e) = \mathit{ld}$ and $o(e') = \mathit{ld}]$ This case is not possible by definition of $\to_c$.
  - $[e \in W(E, a)$ and $o(e') = \mathit{ld}$ and $c(e') = c(e)]$. In the execution of e' it can not be the case that `ml[p,a] == ` $c(e')$ because $e \notin E(p(e'))$ by definition of $\to_{\mathit{rhb}}$, and no other store could have written that value to `ml[p,a]` (by E2). Thus it must be the case that `m[a] == ` $c(e')$ which implies $\mathit{commit}(e) < e' = \mathit{commit}(e')$.
  - $[e \in W(E, a)$ and $o(e') = \mathit{ld}$ and $\mathit{cind}(e') > c(e)]$ In this case there must exist a $e'' \in W(E, a)$ such that $e \to_c e''$ and $e'' \to_c e'$ and $\mathit{cind}(e'') = c(e')$, and we can apply the respective subcases.

This concludes the proof of (TSO1). It remains to show that (TSO2) holds. Suppose $e \to_p e'$. Then as before, $e, e' \in E(p)$ for some p, and $a(e) = a(e') = a$ for some $a \in \mathit{Adr}$. Furthermore, $e \to_p e'$ implies $e < e'$ because `i[p]` is updated monotonically. Assume $e' \to_c e$. The following case distinction shows that a contradiction results, which proves (TSO2) and concludes the proof of $\mathcal{T}(A_{\mathit{TSO}}) \subseteq \mathcal{T}^{\mathit{TSO}}$.

- $[e \in S(E)$ and $e' \in L(E)]$. First, observe that `ml[p,a]` grows monotonically because the stores commit in the order issued. Now, because the execution of e' reads `ml[p,a]`, the assumption $e' \to_c e$ implies $e' < e$ which is a contradiction.
- $[e \in L(E)$ and $e' \in S(E)$ and $c(e) = c(e')]$. Then the execution of e must satisfy either `ml[p,a] == ` $c(e)$ or `m[a] == ` $c(e)$. In either case, it follows that $e' < e$ which is a contradiction.
- $[e \in L(E)$ and $e' \in S(E))$ and $c(e') < c(e)]$ In this case, there must exist a $e'' \in W(E, a)$ such that $e' \to_c e''$ and $e'' \to_c e$ and $\mathit{cind}(e'') = c(e)$. If $e' \notin E(p)$, a $\to_{\mathit{rhb}}$-cycle results and (TSO1) is violated, which is a contradiction. Otherwise, it must be the case that $e' \to_p e''$ (because stores commit in the order issued) and we can apply the previous case.
- [Other cases] In all other cases, a $\to_{\mathit{rhb}}$-cycle results and (TSO1) is violated, which is a contradiction.

We now proceed to the second half of the proof and show $\mathcal{T}^{\mathit{TSO}} \subseteq \mathcal{T}(A_{\mathit{TSO}})$. Now, suppose we are given a trace $E \in \mathcal{T}^{\mathit{TSO}}$. Then we define the transition set $E' = \mathit{issue}(E) \cup \mathit{commit}(E)$ (note that this is not a disjoint union because $\mathit{issue}(e) = \mathit{commit}(e) = e$ for events $e \notin S(E)$). Now, let $\to$ be the least binary relation on E' such that the following conditions are satisfied for all $e_1, e_2 \in E$:

$(R1)$ If $e_1 \to_{\mathit{rhb}} e_2$ then $\mathit{commit}(e_1) \to \mathit{commit}(e_2)$
$(R2)$ If $e_1 \to_p e_2$ then $\mathit{issue}(e_1) \to \mathit{issue}(e_2)$
$(R3)$ If $\mathit{commit}(e_1) \neq e_1$ then $e_1 \to \mathit{commit}(e_1)$

We now claim that $\to$ is acyclic on E', by showing that a cycle leads to a contradiction. Let $e_1 \to \ldots \to e_{k-1} \to e_k = e_1$ be a minimal cycle in E'. Then $k \geq 2$ because none of the rules (R1),(R2),(R3) introduce self-loops. Now, distinguish the following cases to conclude that $\to$ is acyclic on E':

- $[o(e_i) \neq \mathit{st}$ for all $i]$. This implies that we can find $e'_i \in E$ such that $e_i = \mathit{commit}(e'_i)$. Now, $e_i \to e_{i+1}$ implies $e'_i \to_{\mathit{rhb}} e'_{i+1}$ (because the $\to$ edge was either produced by (R1), or it was produced by (R2) in which case $e'_i \to_p e'_{i+1}$ implies $e'_i \to_{\mathit{rhb}} e'_{i+1}$ because $o(e'_i) \neq \mathit{st}$). Thus the e'_i form a $\to_{\mathit{rhb}}$ cycle on E which contradicts (TSO1).
- $[o(e_i) = \mathit{st}$ for some i $]$. Without loss of generality, we assume that the number of i such that $o(e_i) = \mathit{st}$ is minimal, and that $o(e_1) = \mathit{st}$. Now, it must be the rule (R2) that produces the edge $e_{k-1} \to e_1$ (because neither (R1) nor (R3) match $\mathit{op}(e_1) = \mathit{st}$). Thus, there exists a $x \in E$ such that $\mathit{issue}(x) = e_{k-1}$ and $x \to_p e_1$. This means it can not be (R2) that produces $e_1 \to e_2$ (otherwise we can shorten the cycle, because $\to_p$ is transitive). Clearly (R1) does not match either, so $e_1 \to e_2$ must be produced by (R3). Therefore $e_2 = \mathit{commit}(e_1)$. Now, $x \to_p e_1$ implies $x \to_{\mathit{rhb}} e_1$, which implies $\mathit{commit}(x) \to \mathit{commit}(e_1) = e_2$. Now, if it is the case that $\mathit{issue}(x) = \mathit{commit}(x)$, then $e_{k-1} = \mathit{commit}(x)$ and thus $e_{k-1} \to e_2$ so we can shorten the cycle which is a contradiction. On the other hand, if $\mathit{issue}(x) \neq \mathit{commit}(x)$, then $x = \mathit{issue}(x) = e_{k-1}$ and $x \neq \mathit{commit}(x)$, so we apply (R3) to get $e_{k-1} \to \mathit{commit}(x) \to \mathit{commit}(e_1) = e_2$. Thus, we have found a cycle with one less store operation which contradicts minimality.

Now, let $w = w_0 \ldots w_n \in T^*_{\mathit{TSO}}$ be a sequence of pairwise distinct transitions such that $\{w_0, \ldots, w_n\} = E'$ and such that $i < j$ whenever $w_i \to w_j$ (such a sequence always exists because E' is finite and $\to$ is acyclic). We then claim that w is an accepted run by A_{TSO}. To show that this is indeed true, consider the state sequence $s_0, s_1, \ldots, s_n$ such that $\mathit{initial}_{\mathit{TSO}}(s_0) = \mathit{true}$ and $\mathit{apply}_{\mathit{TSO}}\langle w_i \rangle(s_i) = s_{i+1}$. Then it remains to show that (1) for all i, $\mathit{guard}_{\mathit{TSO}}\langle w_i \rangle(s_i) = \mathit{true}$, and (2) $\mathit{accept}_{\mathit{TSO}}(s_n) = \mathit{true}$. We do this by examining the conditions that appear in these expressions individually.

- The condition `i[p]==i-1` (which appears in the guards of events in $E(p)$) is guaranteed to be satisfied because `i[p]` starts as zero and is incremented by each $w_i \in E(p)$. Now, (R2) guarantees that the latter are applied in order.
- The condition `m[a]==c-1` (which appears in the guards of events in $W(E, a)$) is guaranteed to be satisfied because `m[a]` starts as zero and is incremented by each $w_l \in W(E, a)$. Now, (R1) guarantees that the latter are applied in order.
- The condition `B[p].length()==0` appearing in $\mathit{accept}_{\mathit{TSO}}$ is satisfied in the final state for all p because (1) the queues are initially empty, (2) the only transitions that increment/decrement the queue size are $\mathit{st}(p, i, a, c)$ and $\mathit{st}^c(p, i, a, c)$, respectively, and (3) by construction, $E' = \mathit{issue}(E) \cup \mathit{commit}(E)$ and

we can thus pairwise match the transitions $\mathit{st}(p,i,a,c)$ and $\mathit{st}^c(p,i,a,c)$ within E', and by (R3) we know that for each such pair, $\mathit{st}(p,i,a,c)$ precedes $\mathit{st}^c(p,i,a,c)$.

- The condition `B[p].length()==0` appearing in the guards of interlocked events by processor p is satisfied because as before, the only transitions that increment/decrement the queue size are *st* and st^c events by p. But thos are guaranteed to precede the interlocked event, by (R1).
- The condition `B[p].peek()==st(p,i,a,c)` appearing in the guard of $\mathit{st}^c(p,i,a,c)$ is satisfied because (R3) guarantees that $\mathit{st}(p,i,a,c)$ has been added to the FIFO buffer, and (R1) guarantees that we pop the elements in order.
- The condition `c==max{m[a],ml[p,a]}` which appears in the guards of load events is guaranteed for the following reason. Let $l = \mathit{ld}(p,i,a,c) \in E$ be the load event in question. Note that `m[a]` and `ml[p,a]` are only modified by events in $W(E,a)$. Now, by (E3), either $c = 0$, or there exists an event $\mathit{st}(p',i',a,c) \in E$. If $c = 0$, then for all $s \in W(E,a)$, we have $l \rightarrow_{\mathit{rhb}} s$, which implies that l precedes $commit(s)$ (by (R1)), and that l precedes $issue(s)$ if $p(s) = p$. Thus `m[a]` and `ml[p,a]` still have the initial value 0, and the condition is satisfied. On the other hand, if there is an event $s = \mathit{st}(p',i',a,c) \in E$, then distinguish the following two cases.
  - $[p \neq p']$ Then $s \rightarrow_{\mathit{rhb}} l$, and all other writes $s' \in W(E,a)$ must satisfy either $s' \rightarrow_c s$ (meaning that $commit(s')$ precedes $commit(s)$ by (R1)), or $l \rightarrow_c s'$ (meaning that $commit(s')$ trails $commit(l)$ by (R1)). Thus `m[a]` $= c$ when l executes. Now, it remains to show that `ml[p,a]` $<= c$. This follows trivially if `ml[p,a]` has its initial value 0; otherwise, suppose there is a store $s' = \mathit{st}(p,i'',a,c')$ with $c' > c$ that writes to `ml[p,a]` before l. But then $l \rightarrow_c s'$ which means not $s' \rightarrow_p l$ by (TSO2), thus $l \rightarrow_p s'$ and thus $issue(l)$ precedes $issue(s')$ which is a contradiction.
  - $[p = p']$ Then $s \rightarrow_{\mathit{rhb}} l$, and all other writes $s' \in W(E,a) \cap E(p)$ must satisfy either $s' \rightarrow_c s$ (implying $s' \rightarrow_p s$ by (TSO2) and thus $issue(s')$ precedes $issue(s)$ by (R2)), or $l \rightarrow_c s'$ (implying $l \rightarrow_p s'$ by (TSO2) and thus $issue(l)$ precedes $issue(s')$ by (R2)). Thus `ml[p,a]` $= c$ when l executes. Now, it remains to show that `m[a]` $<= c$. This follows trivially if `m[a]` has its initial value 0; otherwise, suppose there is a store $s' = \mathit{st}(p'',i'',a,c')$ with $c' > c$ that writes to `m[a]` before l. But then $l \rightarrow_c s'$ which means $l \rightarrow_{\mathit{rhb}} s'$ which in turn implies that $commit(l)$ precedes $commit(s')$ by (R1), which is a contradiction.

□