Effective Program Verification for Relaxed Memory Models

Sebastian Burckhardt and Madanlal Musuvathi

Microsoft Research

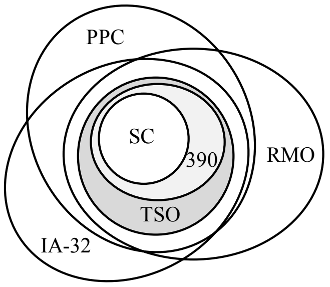
Abstract. Program verification for relaxed memory models is hard. The high degree of nondeterminism in such models challenges standard verification techniques. This paper proposes a new verification technique for the most common relaxation, store buffers. Crucial to this technique is the observation 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 two previously unknown bugs 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 [4,20]. 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 [18], 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 [4,13,22,25] but scalability and automation remain a challenge.

A. Gupta and S. Malik (Eds.): CAV 2008, LNCS 5123, pp. 107–120, 2008.



(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)

Fig. 1. (a) A comparison of various memory models [6,9,14,15,24]. (b) An execution that is possible on *TSO* but not on *SC*.

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) [24], 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 2.

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 1). 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 [21] 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 programs and we resort to iterative context-bounding [21], which provides verification guarantees up to a specific preemption bound.

Section 4 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 [17]. 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 two store buffer problems that the developer immediately agreed were real errors. These bugs were never detected during the extensive code-review and testing the component underwent.

Related Work. Prior work has addressed the verification of programs for relaxed memory models using explicit state enumeration [7,13,22] and using constraint solving [3,4,11,26]. 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 [13] 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 [1,12,18]. The Java Memory Model [19] 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 [8,23] 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.

2 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.

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:

$$\begin{aligned}
\text{(commands issued by processor } p\text{)}\ & E(p) = \{e \in E \mid p(e) = p\} \\
\text{(load events)}\ & L(E) = \{e \in E \mid o(e) = ld\} \\
\text{(store events)}\ & S(E) = \{e \in E \mid o(e) = st\} \\
\text{(events that write to memory)}\ & W(E) = \{e \in E \mid o(e) \in \{st, il\}\} \\
\text{(events that read from memory)}\ & R(E) = \{e \in E \mid o(e) \in \{ld, il\}\} \\
\text{(events that write location } a\text{)}\ & W(E, a) = \{e \in W(E) \mid a(e) = a\}
\end{aligned}$$

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 $E \subseteq E'$.*

[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.

To reason about traces, we introduce binary relations $\rightarrow_p$ and $\rightarrow_c$:

- We use the program order $\rightarrow_p \subseteq \mathit{Evt} \times \mathit{Evt}$ to describe the relative order of events by the same processor. Specifically, we define $e \rightarrow_p e'$ if and only if $p(e) = p(e')$ and $i(e) < i(e')$. For any trace E, $\rightarrow_p$ is a partial order on E and a total order on $E(p)$ for all $p \in \mathit{Proc}$.
- We use the conflict order $\rightarrow_c \subseteq \mathit{Evt} \times \mathit{Evt}$ to describe the relative order of conflicting accesses (where we call two accesses $e, e' \in \mathit{Evt}$ *conflicting* if $a(e) = a(e')$ and $\{e, e'\} \cap W(\mathit{Evt}) \neq \emptyset$). Specifically, we define: $e \rightarrow_c e'$ if and only if $a(e) = a(e')$ and either (1) $o(e') \in W(\mathit{Evt})$ and $c(e) < c(e')$, or (2) $(e, e') \in W(\mathit{Evt}) \times L(\mathit{Evt})$ and $c(e) \leq c(e')$. The conflict order is not actually an 'order' in the mathematical sense because it is not transitive.

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* $\rightarrow_{rhb}$:

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

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. For a detailed proof that Definitions 2 and 3 are equivalent to more intuitive operational descriptions, we refer to our technical report [5].

We now formally define the set of traces $\mathcal{T}_\pi^Y$ that a program π may exhibit on a memory model $Y \in \{SC, TSO\}$. To keep our formalization light, we represent a program π abstractly by a function $next_\pi : \mathcal{T} \times Proc \rightarrow \mathcal{P}(Op \times Adr)$. 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

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

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

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

3 Solution

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

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

Theorem 1. *A program π is store buffer safe if and only if it has no borderline traces.*

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

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 1 (Cycle Characterization). *Let $E \in \mathcal{T}_\pi^{SC}$ 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) $E' \notin \mathcal{T}_\pi^{SC}$ if and only if $o = ld$ and $write(E, a, c+1) \rightarrow_{hb}^{*} last(E, p)$.
(2) $E' \notin \mathcal{T}_\pi^{TSO}$ if and only if $o = ld$ 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 $o = ld$ 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 $E' \notin \mathcal{T}_\pi^{SC}$. **(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 $E' \notin \mathcal{T}_\pi^{TSO}$. 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 $E' \notin \mathcal{T}_\pi^{TSO}$. **(1⇒).** Assume $E' \notin \mathcal{T}_\pi^{SC}$. 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 $o = ld$ (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 $o(e_1) \in \{st, il\}$, $a(e_1) = a$ 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 $E \in \mathcal{T}_\pi^{SC}$), thus $e_n \rightarrow_p e$. Therefore, either $e_n = last(E, p)$ or $e_n \rightarrow_p last(E, p)$. We can thus conclude that $write(E, a, c+1) \rightarrow_{hb}^{*} last(E, p)$ as desired. **(2⇒).** If $E' \notin \mathcal{T}_\pi^{TSO}$ then $E' \notin \mathcal{T}^{TSO}$ (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 $\rightarrow_{rhb} \subseteq \rightarrow_{hb}$), we conclude that there exists a cycle of the form $e \rightarrow_{rhb} e_1 \rightarrow_{rhb} \ldots \rightarrow_{rhb} e_n \rightarrow_{rhb} e$, that $e \rightarrow_c e_1$, that $o = ld$, that $write(E, a, c+1) \rightarrow_{rhb}^{*} e_n$, and that $e_n \rightarrow_p e$. The latter implies that $o(e_n) \neq st$ (otherwise not $e_n \rightarrow_{rhb} e$), and therefore either $e_n = lastR(E, p)$ or $e_n \rightarrow_{rhb} lastR(E, p)$. 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 $o(e) = ld$ (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 $o(e') \in \{st, il\}$, $a(e') = a$ and $c(e') > c$. Thus, condition (ii) is satisfied with $c' = c(e')$.

```
1  function is_store_buffer_safe($e_1 e_2 \ldots e_n$) returns boolean {
2    var k,p,a,c : $\mathbb{N}$; var E : $\mathcal{T}$;
3    E := $\emptyset$;
4    for (k := 1; k <= n; k++) {
5      if ($o(e_k)$ = ld) {
6        p := $p(e_k)$; a := $a(e_k)$; c := $c(e_k)$;
7        while (c > 0) {
8          if (p = $i$($write$(E,a,c)))
9            break;
10         if ($write$(E,a,c) $\to^*_{rhb}$ $lastR$(E,p))
11           break;
12         if ($write$(E,a,c) $\to^*_{hb}$ $last$(E,p))
13           return false;
14         c := c - 1;
15       }
16     }
17     E := E $\cup$ $e_k$;
18   }
19   return true;
20 }
```

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

3.1 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 $w = e_1 \ldots e_n \in Evt^*$ of events, let $E_w = \{e_1, \ldots e_n\}$. The sequence w is called an *interleaving* of a program π if (1) the e_k are pairwise distinct, (2) $E_w \in \mathcal{T}_\pi^{SC}$, (3) $e_x \to_{hb} e_y \implies x < y$, and (4) $next_\pi(E_w, p) = \emptyset$ for all $p \in Proc$.

Theorem 2 (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 $w = e_1 \ldots e_n$. 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 $E = \{e_1, \ldots, e_{k-1}\}$, and $e_k = ld(p, i, a, c)$ for some c. Let $e = e_k$, and let $e' = ld(p, i, a, c' - 1)$. We now argue that $E' = E \cup \{e'\} \in (\mathcal{T}_\pi^{TSO} \setminus \mathcal{T}_\pi^{SC})$, which implies that E is a borderline trace and thus $\mathcal{T}_\pi^{SC} \neq \mathcal{T}_\pi^{TSO}$ by Theorem 1 as desired. First, note that $e' \in succ_\pi(E)$ because $E \cup \{e\} \in \mathcal{T}_\pi^{SC}$ implies $E \cup \{e'\} \in \mathcal{T}$ and $(o, a) \in next_\pi(E, p)$ (using that π is locally deterministic). We can thus enlist the help of Lemma 1 to show $E' \in (\mathcal{T}_\pi^{TSO} \setminus \mathcal{T}_\pi^{SC})$. First, because the program returned at line 13, we know $write(E, a, c') \to^*_{hb} last(E, p)$, which implies $E' \notin \mathcal{T}_\pi^{SC}$ by Lemma 1, part (1). Second, because the program did not break at line 11 right before returning on line 13, we know that not $(write(E, a, c') \to^*_{rhb} lastR(E, p))$. Moreover, because the while loop was not broken at line 9, we know that $p(\mathit{write}(E, a, c'')) \neq p$ for all $c'' \geq c'$. By Lemma 1, part (2) we conclude that $E' \in \mathcal{T}_\pi^{TSO}$.

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 $I \subseteq \mathit{Evt}^*$ *representative* for program π if for all $E \in \mathcal{T}_\pi^{SC}$ there exists an interleaving $w \in I$ such that $E \subseteq E_w$ and there are no $\rightarrow_{hb}$-edges from $E_w \setminus E$ into E.

Theorem 3 (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 1, we know that $\mathcal{T}_\pi^{SC} \neq \mathcal{T}_\pi^{TSO}$ implies that there exists a borderline trace $E \in \mathcal{T}_\pi^{SC}$. Thus there exists an element $e = o(p, i, a, c) \in \mathit{Evt}$ such that $E' = (E \cup \{e\}) \in \mathcal{T}_\pi^{TSO} \setminus \mathcal{T}_\pi^{SC}$. 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 1, part (1), we know that $\mathit{write}(E, a, c{+}1) \rightarrow_{hb}^* \mathit{last}(E, p)$ 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 $\rightarrow_{hb}$), this implies $\mathit{write}(E_k, a, c{+}1) \rightarrow_{hb}^* \mathit{last}(E_k, p)$ 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 13 unless it is broken at either line 9 or line 11. But that is not possible, for the following reasons. First, suppose line 9 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 $E' \notin \mathcal{T}_\pi^{TSO}$ by Lemma 1, part (2ii), contradicting the assumption. Next, suppose line 11 breaks. Let c' be the value of the variable c at that time; then $c{+}1 \leq c' \leq c(e_k)$ and $\mathit{write}(E_k, a, c') \rightarrow_{rhb}^* \mathit{lastR}(E_k, p)$ within E_k. Now, because $E(p) = E_k(p)$, $\mathit{lastR}(E_k, p) = \mathit{lastR}(E, p)$. Because there are no $\rightarrow_{hb}$-edges (and thus no $\rightarrow_{rhb}$-edges) from E_w into E, this implies that $\mathit{write}(E, a, c') \rightarrow_{rhb}^* \mathit{lastR}(E, p)$. Because $c + 1 \leq c'$, this implies $\mathit{write}(E, a, c) \rightarrow_{rhb}^* \mathit{lastR}(E_k, p)$, which in turn implies $E' \notin \mathcal{T}_\pi^{TSO}$ by Lemma 1, part (2i), contradicting the assumption.

A stateless model checker (such as Verisoft [10] or CHESS[21]) 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 $E \in \mathcal{T}_\pi^{SC}$). The following theorem (proved in [5]) clarifies that this is true even if partial order reduction is employed. We call a set of interleavings $I \subseteq \mathit{Evt}^*$

```
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;
}
```

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

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

3.2 Vector Clocks

The pseudocode in Fig. 2 does not detail how to decide the conditions on lines 10 and 12. 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. 7 below) and by engineering a vector clock instance (Fig. 3) that can compute the transitive closure $\rightarrow^*_{rhb}$ in time $O(nN^2)$.

Theorem 5. *Let $w = e_0 \dots e_n$ be an interleaving of some program π, and let $t_1, \dots, t_n$ 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 $k \in \{1, \dots, 2N\}$.*

We now describe informally how this vector clock works (for a detailed proof of the theorem see [5]). 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 7 (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, G is a set of groups, in, out are functions $\Sigma \rightarrow \mathcal{P}(C)$, and gps is a function $\Sigma \rightarrow \mathcal{P}(G)$ such that the following conditions are satisfied:*

(VC1) for all $\sigma \in \Sigma$, $gps(\sigma) \neq \emptyset$.
(VC2) for all $g \in G$, $\rightarrow$ is a total order on $\{\sigma \in \Sigma \mid g \in gps(\sigma)\}$.
(VC3) for all $\sigma, \sigma' \in \Sigma$, we have $(out(\sigma) \cap in(\sigma') \neq \emptyset) \Leftrightarrow (\sigma \rightarrow \sigma')$.

4 Experiments

We present experimental results for four C# programs (Fig. 4(a)). 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 [21] (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 [21]; 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.

Figure 4(b) describes a memory model bug that we found in a production level concurrency library at Microsoft [5]. The program uses two flags `isIdling` and `hasWork` as well as a condition variable to synchronize between consumers and producers. An idle consumer waits on the condition variable if `hasWork` is

program	context	# interleavings		time	ver. time [s]	
name	bound	total	borderline	[s]	SoBeR	CHESS
Fig. 1(b)	∞	10	4	< 0.1	< 0.2	< 0.2
dekker	1	5	4	< 0.1	< 0.2	< 0.2
(2 threads,	2	36	23	< 0.1	0.39	0.37
2 crit-sec)	3	183	50	< 0.1	1.9	1.8
(loc 82)	4	1,219	124	< 0.1	13.2	13.0
	5	8,472	349	< 0.1	106.0	100.6
bakery	0	1	1	< 0.1	< 0.2	< 0.2
(2 threads,	1	25	20	< 0.1	0.47	0.43
3 crit-sec)	2	742	533	< 0.1	10.3	9.8
(loc 122)	3	12,436	8,599	< 0.1	189.0	181.0
takequeue	0	3	0	n.a.	< 0.3	< 0.3
(2 threads,	1	47	14	0.34	0.72	0.69
6 ops)	2	402	189	0.43	5.2	4.9
(loc 374)	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

(a)

```
volatile bool isIdling;
volatile bool hasWork;

//Consumer thread
void BlockOnIdle(){
   lock (condVariable){
      isIdling = true;
      if (!hasWork)
          Wait(condVariable);
      isIdling = false;
   }
}

//Producer thread
void NotifyPotentialWork(){
   hasWork = true;
   if (isIdling)
     lock (condVariable) {
        Pulse(condVariable);
     }
}
```

(b)

Fig. 4. (a) Experiments on a 2.2GHz Intel Core Duo laptop running Windows Vista. **(b)** An example of a store buffer safety bug we found in a production-level C# program.

false, but only after setting `isIdling` to true. To optimize for the common case in which there are no idle consumers, the producer acquires the lock only when `isIdling` is true. Also, to account for a possible race on the `isIdling` flag, the producer sets `hasWork` to true before checking the `isIdling` flag. We can see that in all sequentially consistent executions the producer correctly wakes up the idle consumer, if any. However, in the presence of store buffers, a store can be delayed past a subsequent load.[2] In particular, the consumer can read `hasWork` before its write to `isIdling` is visible to the producer. Thus, the producer may erroneously believe that no consumer is idling, not perform a signal, and leave the consumer waiting forever.

5 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.

[2] Note that unlike Java, the C# memory model does not guarantee a sequentially consistent ordering of volatile accesses.

References

1. Adve, S., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer 29(12), 66–76 (1996)
2. Ben-Ari, M.: Principles of Concurrent Programming. Prentice Hall, Englewood Cliffs (1982)
3. Burckhardt, S., Alur, R., Martin, M.: Bounded verification of concurrent data types on relaxed memory models: A case study. In: Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 489–502. Springer, Heidelberg (2006)
4. Burckhardt, S., Alur, R., Martin, M.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21 (2007)
5. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. Technical Report MSR-TR-2008-12, Microsoft Research (2008)
6. Compaq Computer Corporation. Alpha Architecture Reference Manual, 4th edn. (January 2002)
7. Dill, D., Park, S., Nowatzyk, A.: Formal specification of abstract memory models. In: Symposium on Research on Integrated Systems, pp. 38–52. MIT Press, Cambridge (1993)
8. Fang, X., Lee, J., Midkiff, S.: Automatic fence insertion for shared memory multiprocessing. In: ICS, pp. 285–294 (2003)
9. Frey, B.: PowerPC Architecture Book v2.02. IBM Corporation (2005)
10. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL 1997: Principles of Programming Languages, pp. 174–186 (1997)
11. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or not QB: An efficient execution verification tool for memory orderings. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 401–413. Springer, Heidelberg (2004)
12. Hill, M.: Multiprocessors should support simple memory-consistency models. IEEE Computer 31(8), 28–34 (1998)
13. Huynh, T., Roychoudhury, A.: A Memory Model Sensitive Checker for C#. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 476–491. Springer, Heidelberg (2006)
14. IBM Corporation. z/Architecture Principles of Operation, 1st edn. (2000)
15. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A (November 2006)
16. Intel Corporation. Intel 64 Architecture Memory Ordering White Paper (August 2007)
17. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Communications of the ACM 17(8), 453–455 (1974)
18. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comp. C-28(9), 690–691 (1979)
19. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: Principles of Programming Languages (POPL), pp. 378–391 (2005)
20. Morrison, V.: Understand the impact of low-lock techniques in multithreaded apps. MSDN Magazine 20(10) (October 2005)
21. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
22. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: SPAA, pp. 34–41 (1995)

23. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)
24. Weaver, D., Germond, T.: The SPARC Architecture Manual Version 9. PTR Prentice Hall, Englewood Cliffs (1994)
25. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Memory-model-sensitive data race analysis. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 30–45. Springer, Heidelberg (2004)
26. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: IPDPS (2004)