Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models

Jabob Burnim, Koushik Sen, and Christos Stergiou
{jburnim,ksen,chster}@cs.berkeley.edu
EECS Department, University of California, Berkeley

Abstract. We present a technique for verifying that a program has no executions violating sequential consistency (SC) when run under the relaxed memory models Total Store Order (TSO) and Partial Store Order (PSO). The technique works by monitoring sequentially consistent executions of a program to detect if similar program executions could fail to be sequentially consistent under TSO or PSO. We propose novel monitoring algorithms that are sound and complete for TSO and PSO—if a program can exhibit an SC violation under TSO or PSO, then the corresponding monitor can detect this on some SC execution. The monitoring algorithms arise naturally from the operational definitions of these relaxed memory models, highlighting an advantage of viewing relaxed memory models operationally rather than axiomatically. We apply our technique to several concurrent data structures and synchronization primitives, detecting a number of violations of sequential consistency.

1 Introduction

Programmers writing concurrent software often assume that the underlying memory model is sequentially consistent. However, sequential consistency strongly constrains the ordering of memory operations, which can make it difficult to achieve high performance in commodity microprocessors [9, 20]. Thus, to enable increased concurrency and performance, processors often provide a *relaxed* memory model. Unfortunately, working with relaxed memory models often requires subtle and difficult reasoning [9, 20].

Nevertheless, developers of high-performance concurrent programs, such as lock-free data-structures and synchronization libraries, often use regular load and store operations, atomic compare-and-swap-like primitives, and explicit data races instead of locks to increase performance. Concurrency bugs are notoriously hard to detect and debug; relaxed memory models make the situation even worse.

Recently, there has been great interest in developing techniques for the verification and analysis of concurrent programs under relaxed memory models [9, 20, 13, 3, 15, 4, 2, 10]. In a promising and practical approach for such verification, Burckhardt and Musuvathi [4] argued that programmers, despite using ad-hoc synchronization, expect their program to be sequentially consistent. They proposed SOBER, which monitors sequentially consistent executions to detect violations of sequential consistency (SC). A key observation made in their work is that, for the Total Store Order (TSO) [23] memory model (which is quite similar to that of the x86 architecture [21]), if a program execution under TSO violates sequential consistency (SC), then this fact can be detected by examining some sequentially consistent execution of the program. Therefore, if run-time monitoring is combined with a traditional model checker, which explores all sequentially consistent executions of the program, then all violations of SC under TSO can be detected. Burckhardt and Musuvathi [4] use an axiomatic definition of SC and TSO to derive the SOBER monitoring algorithm.

In this paper, we develop two novel monitoring algorithms for detecting violations of sequential consistency (SC) under relaxed memory models Total Store Order (TSO) [23] and Partial Store Order (PSO) [23]. Each algorithm, when monitoring a sequentially consistent execution of a program, simulates a similar TSO or PSO execution, reporting if this similar execution can ever violate sequential consistency. We prove both monitors *sound*—if they report a warning, then the monitored program can violate SC under TSO or PSO—and *complete*—if a program can violate SC under TSO or PSO, then the corresponding monitor can detect this fact by examining some sequentially consistent execution.

Rather than working with axiomatic definitions of these relaxed memory models, as [4] does, we derive our algorithms from *operational* definitions of TSO and PSO. We show that this alternate approach naturally leads to fundamentally different monitoring algorithms, with several advantages over SOBER.

One advantage of our operational approach is that our monitoring algorithms follow simply from the operational definitions of TSO and PSO. While monitoring algorithms based on axiomatic definitions require the design of complex vector clocks, in addition to the standard vector clocks to track the traditional happens-before relation, our approach can directly "run" the operational memory model. Thus, we were easily able to develop a monitoring algorithm for PSO, in addition to TSO—such an extension is unknown for the SOBER algorithm.

Another advantage of our algorithms is that they have a run-time complexity of $O(N \cdot P)$ when monitoring a sequentially consistent execution with N shared memory operations and P processes. This complexity is an improvement over the run-time complexity $O(N \cdot P^2)$ of the SOBER algorithm. We see this improvement in run-time complexity because we do not need to maintain additional vector clocks for TSO and PSO.

Further, in developing our monitoring algorithms, we discovered a bug in the SOBER algorithm—in its axiomatic definition of TSO—which makes the algorithm incomplete. We believe that this bug is quite subtle. The same bug is also present [21] in the 2007 Intel® 64 Architecture Memory Order White Paper. In this paper, we identify and correct the error [8] in the SOBER algorithm arising from a too-strict axiomatic definition of the TSO memory model.

We have implemented our monitoring algorithms for C programs in THRILLE [16] and, combined with THRILLE's preemption-bounded model checking [19], have applied our monitors to two mutual exclusion algorithms and several concurrent data structures. Our experiments show that we can detect sequential consistency violations under TSO or PSO in all of these benchmarks.

Other Related Work. There have been many efforts to verify or check concurrent programs on relaxed memory models [9, 20, 13, 3, 15, 4, 2, 10]. Some of these techniques [13, 3] encode a program and the underlying memory model as a constraint system and use a constraint solver to find bugs. Other techniques [9, 20, 15] explicitly explore the state space of a program to find bugs.

Recently, [10] proposed an adversarial memory for testing at run-time if a data race in a Java program could be harmful under Java's memory model.

2 Preliminaries

We consider a parallel program P to consist of a number of *parallel threads*, each with a thread-local state, that communicate though a *shared memory.* The *execution* of a parallel program consists of a sequence of program *steps.* In each step, one of the program threads *issues* a shared memory *operation*—e.g., a read or write of a shared variable—and then updates its local state based on its current local state and anything returned by the memory operation—e.g., the value read for a shared variable.

Below, we will give operational definitions of relaxed memory models TSO and PSO as abstract machines. These abstract machine definitions are designed to be simple for a programmer to understand and to reason about, not to exactly describe the internal structure or operation of real hardware processors. For example, our operational definitions contain no caches or mechanisms for ensuring cache coherency. At the same time, these operational definitions are faithful in that they allow exactly the program behaviors allowed by more traditional axiomatic definitions of TSO and PSO.

2.1 Programming Model

Let *Proc* be set of all program processes or thread identifiers and *Value* be the set of possible program values. Then, we define the set *Event* of shared memory operations, or *events*, to consist of all:

- Stores $\mathbf{st}(p, a, v)$ by process $p \in \mathit{Proc}$ to address $a \in \mathit{Adr}$ of $v \in \mathit{Value}$.
- Loads $\mathbf{ld}(p, a)$ by process $p \in \mathit{Proc}$ of address $a \in \mathit{Adr}$.
- Atomic operations $\mathbf{atm}(p, a, f)$ by $p \in \mathit{Proc}$ on $a \in \mathit{Adr}$, which store $f(v)$ to address a after reading $v \in \mathit{Value}$ from a, where $f : \mathit{Value} \to \mathit{Value}$.
  This operations models atomic shared memory primitives such as compare-and-swap (CAS), fetch-and-add, test-and-set, etc.[1]

Note that we need not explicitly include a memory fence operation. In our operational models, memory barriers can be simulated by atomic operations $\mathbf{atm}(p, a, f)$, which restrict reordering or delaying of earlier memory ops.

[1] For example, a CAS(a, *old*, *new*) by process p is modeled by $\mathbf{atm}(p, a, f)$, where $f = (\lambda x.\ \mathbf{if}\ x = \mathit{old}\ \mathbf{then}\ \mathit{new}\ \mathbf{else}\ x)$. The CAS "succeeds" when it reads value *old*, and "fails" otherwise.

We denote the process p and address a of an event $e \in \mathit{Event}$ by $p(e)$ and $a(e)$, respectively.

We now formalize programs as independent processes with local state and communicating only via the above shared memory operations. We abstract unnecessary details such as the source language, control flow, and structure of the process-local state. We define a program P to be a tuple $(s_0, \mathit{next}, \mathit{update})$ where:

- Function $s_0 : \mathit{Proc} \to \Sigma$ is the initial program state, mapping process p to its thread-local state $s_0(p) \in \Sigma$, where Σ is the set of all possible local states.
- Partial function $\mathit{next} : \mathit{Proc} \times \Sigma \to \mathit{Event}$ indicates the next *memory operation* or *event*, $\mathit{next}(p, \sigma)$, that process p issues when in local state σ.
  If next is undefined for (p, σ), denoted $\mathit{next}(p, \sigma) = \bot$, then process p is complete. Program P terminates when $\mathit{next}(p, s(p)) = \bot$ for all p.
- Function $\mathit{update} : \mathit{Proc} \times \Sigma \times \mathit{Value} \to \Sigma$ indicates the new local state $\mathit{update}(p, \sigma, v)$ of process p after it receives response v to op $\mathit{next}(p, \sigma)$.

We similarly formalize a memory model MM as a labeled transition system with initial state m_0 and with transitions labeled by events from *Event*, paired with memory model responses. We also allow memory model MM to have a set of labeled internal transitions τ_{MM}, to model any internal nondeterminism of the memory model. Then, an *execution* of program P under memory model MM is a sequence of labeled transitions:

$$(s_0, m_0) \xrightarrow{l_1} (s_1, m_1) \xrightarrow{l_2} \cdots \xrightarrow{l_n} (s_n, m_n)$$

where each transition $(s_{i-1}, m_{i-1}) \xrightarrow{l_i} (s_i, m_i)$ is either labeled by an ordered pair $(e_i, r_i) \in \mathit{Event} \times \mathit{Value}$, in which case:

- $e_i = \mathit{next}(p(e_i), s_{i-1}(p(e_i)))$
- $s_i(p(e_i)) = \mathit{update}(p(e_i), s_{i-1}(p(e_i)), r_i)$, where r_i is the value returned for e_i, which is $\bot$ for stores.
- $s_i(p') = s_{i-1}(p')$ for $p' \neq p(e_i)$
- $m_{i-1} \xrightarrow{(e_i, r_i)} m_i$ is a transition in MM

or is labeled by an internal transition from τ_{MM}, in which case:

- $s_i = s_{i-1}$, and $m_{i-1} \xrightarrow{l_i} m_i$ is a transition in MM

In this model, there are two sources of nondeterminism in a program execution: (1) The thread schedule—i.e., at each step, which process p executes a transition, and (2) the internal nondeterminism of the memory model.

3 Operational Memory Models

We now give our operational definitions for three memory models: sequential consistency (SC) and relaxed memory models TSO and PSO. Fundamentally, these definitions are equivalent to operational definitions given by other researchers for SC [14, 5, 2, 18], TSO [14, 5, 2, 21, 18], and PSO [14, 2, 18].

In our presentation, we aim for definitions that provide a simple and easy to understand model for a programmer. We present each memory model as a library or module with an internal state, representing the abstract state of a shared memory, and with methods $\mathbf{store}(p, a, v)$, $\mathbf{load}(p, a)$, and $\mathbf{atomic}(p, a, f)$ by which a program interacts with the shared memory. The memory model executes all such methods atomically—i.e. one at a time and uninterrupted. Additionally, memory models TSO and PSO each have an *internal* method $\mathbf{store}^c$. Each memory model is permitted to nondeterministically call this internal method, with any arguments, whenever no other memory model method is being executed.

A note about connecting these definitions to our formalism in Section 2:

- $m \xrightarrow{\mathbf{ld}(p,a),v} m$ when $\mathbf{load}(p, a)$, run in memory model state m, returns v. (Note that $\mathbf{load}(p, a)$ does not modify the memory model state m.)
- $m \xrightarrow{\mathbf{st}(p,a,v),\bot} m'$ when $\mathbf{store}(p, a, v)$, run in state m, yields state m'.
- $m \xrightarrow{\mathbf{atm}(p,a,f),v} m'$ when $\mathbf{atomic}(p, a, f)$, run in m, returns v and yields m'.

Sequential Consistency (SC) Our operational definition of SC [17] is given in Figure 1. The SC abstract machine simply models shared memory as an array m mapping addresses to values, reading from and writing to this array on loads, stores, and atomic operations.

Total Store Order (TSO) Our definition of TSO [23] is given in Figure 2. In addition to modeling shared memory as array m, mapping addresses to values, the TSO abstract machine has a FIFO *write buffer* $B[p]$ for each process p.

We omit a proof that our operational definition is equivalent to more traditional axiomatic ones. Our model is similar to the operational definitions in [4] and [21], both of which are proved equivalent to axiomatic definitions of TSO. Conceptually, the per-process write buffers allow stores to be reordered or delayed past later loads, while ensuring that each process's stores become globally visible in the order in which they are performed. And there is a total order on all stores—the order in which stores *commit*—that respects the program order.

Partial Store Order (PSO) Our operational definition of PSO [23] is given in Figure 3. Our PSO abstract machine is very similar to that of TSO, except that pending writes are stored in per-process and *per-address* write buffers. Internal method $\mathbf{store}^c_{PSO}(p, a)$ commits the oldest pending store *to address* a and $\mathbf{atomic}_{PSO}(p, a, f)$ commits/flushes only pending stores *to address* a.

```
m[$Adr$] : $Val$

$\mathbf{store}_{\mathrm{SC}}(p, a, v)$ :
    $m[a] := v$

$\mathbf{load}_{\mathrm{SC}}(p, a)$ :
    return $m[a]$

$\mathbf{atomic}_{\mathrm{SC}}(p, a, f)$ :
    $ret := m[a]$
    $m[a] := f(m[a])$
    return ret
```

Fig. 1. Operational Model of Sequential Consistency (SC).

```
m[$Adr$] : Val
B[$Proc$] : **FIFOQueue** of ($Adr$, $Val$)

**store**$_{\mathrm{TSO}}(p, a, v)$ :
    $B[p].addLast(a, v)$

**store**$^{c}_{\mathrm{TSO}}(p)$ :
    **if not** $B[p].empty()$:
        $(a, v) := B[p].removeFirst()$
        $m[a] := v$

**load**$_{\mathrm{TSO}}(p, a)$ :
    **if** $B[p].contains((a, *))$:
        $(a, v) :=$ last element $(a, *)$ of $B[p]$
        **return** $v$
    **else**:
        **return** $m[a]$

**atomic**$_{\mathrm{TSO}}(p, a, f)$:
    **while not** $B[p].empty()$:
        **store**$^{c}_{\mathrm{TSO}}(p)$
    $ret := m[a]$
    $m[a] := f(m[a])$
    **return** ret
```

Fig. 2. Operational Model of TSO.

```
m[$Adr$] : Val
B[$Proc$][$Adr$] : **FIFOQueue** of $Val$

**store**$_{\mathrm{PSO}}(p, a, v)$ :
    $B[p][a].addLast(v)$

**store**$^{c}_{\mathrm{PSO}}(p, a)$ :
    **if not** $B[p][a].empty()$:
        $m[a] := B[p][a].removeFirst()$

**load**$_{\mathrm{PSO}}(p, a)$ :
    **if not** $B[p][a].empty()$:
        **return** $B[p][a].getLast()$

    **else**:
        **return** $m[a]$

**atomic**$_{\mathrm{PSO}}(p, a, f)$:
    **while not** $B[p][a].empty()$:
        **store**$^{c}_{\mathrm{PSO}}(p, a)$
    $ret := m[a]$
    $m[a] := f(m[a])$
    **return** ret
```

Fig. 3. Operational Model of PSO.

4 Violations of Sequential Consistency

In this section, we formally define what it means for a program to have a *violation* of sequential consistency under a relaxed memory model. In Section 5, we will give monitoring algorithms for detecting such violations under TSO and PSO by examining only the SC executions of a program.

4.1 Execution Traces

If a program exhibits some behavior in a TSO or PSO execution, we would like to say that the behavior is not sequentially consistent if there is no execution under SC exhibiting the same behavior. We will define a *trace* of an SC, TSO, or PSO execution to capture this notion of the behavior of a program execution.

Following [12], [4], and [6], we formally define a trace of an execution of a program P to be a tuple $(E, \rightarrow_p, src, \rightarrow_{st})$, where

- $E \subseteq Event$ is the set of shared memory operations or events in the execution.
- For each process $p \in Proc$, relation $\rightarrow_p \subseteq E \times E$ is a total order on the events $e \in E$ from process p—i.e. with $p(e) = p$. In particular, $e \rightarrow_p e'$ iff $p(e) = p(e')$ and e is before e' in the execution. Thus, $\rightarrow_p$ does not relate events from different processes.

Relation $\rightarrow_p$ is called the *program order relation*, and $e \rightarrow_p e'$ indicates that process p issued operation e before issuing e'.

- For each load or atomic operation $e \in E$, partial function $src : Event \rightarrow Event$ indicates the store or atomic operation $src(e) \in Event$ from which e reads its value. $src(e) = \bot$ indicates that e got its value from no store, instead reading the initial default value in the shared memory. Note that $a(src(e)) = a(e)$ whenever $src(e)$ is defined.
- For each address a, relation $\rightarrow_{st} \subseteq E \times E$ is a total order on the stores and atomic operations on a in the execution. In particular, $e \rightarrow_{st} e'$ iff $a(e) = a(e')$ and e becomes globally visible before e' in the execution. Thus, $\rightarrow_{st}$ does not relate events on different addresses.
  Memory models SC, TSO, and PSO all guarantee the existence, for each address, of such a total order in which the writes to that address become globally visible[2]. Note that not all relaxed memory models guarantee the existence of such an ordering.

Note that we can only define a trace for a *complete* TSO or PSO execution—that is, one in which every store has become globally visible or, in the language of our abstract models, committed. If multiple processes have pending writes to the same address, then the execution does not specify an order on those writes and thus does not define a total $\rightarrow_{st}$ relation.

4.2 Sequential Consistency & The Happens-Before Relation

We define a trace of a program P to be sequentially consistent only if it arises from some sequentially consistent execution of P:

Definition 1. *A trace $T = (E, \rightarrow_p, src, \rightarrow_{st})$ of a program P is* sequentially consistent *iff there exists some execution of P under SC with trace T.*

This definition is not very convenient for showing that a trace is *not* sequentially consistent. Thus, following [22, 4], we give an axiomatic characterization of SC traces by defining relations $\rightarrow_c$ and $\rightarrow_{hb}$ on the events of a trace:

Definition 2. *Let $(E, \rightarrow_p, src, \rightarrow_{st})$ be a trace. Events $e, e' \in E$ are related by the* **conflict-order relation***, denoted $e \rightarrow_c e'$, iff $a(e) = a(e')$ and one of the following holds:*

- *e is a write, e' is a read, and $e = src(e')$,*
- *e is a write, e' is a write, and $e \rightarrow_{st} e'$*
- *e is a read, e' is a write, and either $src(e) = \bot$ or $src(e) \rightarrow_{st} e'$*

Definition 3. *For a trace $(E, \rightarrow_p, src, \rightarrow_{st})$, the* happens-before relation *is defined as the union of the program-order and conflict-order relations on E—i.e. $\rightarrow_{hb} = (\rightarrow_p \cup \rightarrow_c)$. We refer to the reflexive transitive closure of the happens-before relation as $\rightarrow_{hb}^*$.*

[2] This property is closely related to *store atomicity* [1]. TSO and PSO do not technically have store atomicity, however, because a process's loads can see the values of the process's earlier stores before those stores become globally visible.

As observed in [22] and [4], a trace is sequentially consistent iff its $\rightarrow_{hb}$ relation is acyclic:

Proposition 1. *Let $T = (E, \rightarrow_p, src, \rightarrow_{st})$ be a trace of an execution of program P. Trace T is sequentially consistent iff relation $\rightarrow_{hb}$ is acyclic on E.*

We define a sequential consistency *violation* as a program execution with a non-sequentially-consistent trace:

Definition 4. *A program P has a* violation of sequential consistency *under relaxed memory model TSO (resp. PSO) iff there exists some TSO (resp. PSO) execution of P with trace T such that T is not sequentially consistent.*

5 Monitoring Algorithms

We describe our monitoring algorithms for TSO and PSO in this section. We suppose here that we are already using a model checker to explore and to verify the correctness of all sequentially consistent executions of program P. A number of existing model checkers [19, 16] explore the sequentially consistent, interleaved schedules of a parallel program.

Figures 4 and 5 list our monitor algorithms for TSO and PSO. We present our algorithms as *online* monitors—that is, for an SC execution $(s_0, m_0) \xrightarrow{e_1, r_1} \cdots \xrightarrow{e_n, r_n} (s_n, m_n)$ of some program P, we run **monitor**$_{TSO}$ (respectively **monitor**$_{PSO}$) on the execution by:

- Initializing the internal state B and **last** as described in Figure 4 (resp. 5).
- Then, for each step $e_i \in Event$ in the execution, from e_1 to e_n, we call **monitor**$_{TSO}(e_i)$.
- If any of these calls signals "SC Violation", then P has a sequential consistency violation under TSO (resp., PSO). Note that we do not stop our monitoring algorithm after detecting the first violation. Thus, we find all such violations of sequential consistency along each SC execution.

Conceptually, each monitor algorithm works by simulating a TSO (respectively PSO) version of the given SC execution. Array B simulates the FIFO write buffers in a TSO (resp. PSO) execution, but buffers the pending store *events* rather than just the pending values to be written. Lines 16–25 update these write buffers, queueing a store when the SC execution performs a store and flushing the buffers when the SC execution performs an atomic operation.

But when should this simulated TSO (resp. PSO) execution "commit" these pending stores, making them globally visible? Lines 10–12 commit these pending stores as late as possible while still ensuring that the simulated TSO (resp. PSO) execution has the same trace and $\rightarrow_{hb}$-relation as the SC execution. This is achieved by, just before simulating operation e by process p on address a, committing all pending stores to address a from any other process—these pending stores *happen before* e in the SC execution, so they must be committed to ensure they *happen before* e in the simulated TSO (resp. PSO) execution. Note that, in the TSO monitor, this may commit pending stores to other addresses, as well.

```
1  B[$Proc$] : **FIFOQueue** of ($Adr$, $Event$)
2  prev[$Proc$] : $Event$ initialized to $\bot$
3
4  **monitor**$_{TSO}$(e):
5    // *Could simulation have $\rightarrow_{hb}-$cycle?*
6    **if** $\exists(a(e), e') \in B[p(e')]$ **with**
           $p(e') \neq p(e) \wedge e' \rightarrow_{hb}^{*}$prev$[p(e)]$:
7      **signal** "SC Violation"
8
9    // *Ensure equivalence to SC.*
10   **while** $\exists(a(e), *) \in B[p']$ **with** $p' \neq p(e)$:

11     $B[p'].removeFirst()$
12     **emit** $\mathbf{st}^c(p')$
13
14   // *Execute e in TSO simulation.*
15   prev$[p(e)] := e$
16   **if** $e =$**st**$(p, a, v)$:
17     $B[p].addLast(a, e)$
18     **emit** **st**$(p, a, v)$
19   **else if** $e =$**ld**$(p, a)$:
20     **emit** **ld**$(p, a)$
21   **else if** $e =$**atm**$(p, a, f)$:
22     **while not** $B[p].empty()$:
23       $B[p].removeFirst()$
24       **emit** $\mathbf{st}^c(p)$
25     **emit** **atm**$(p, a, f)$
```

Fig. 4. Monitoring algorithm for TSO

```
1  B[$Proc$][$Adr$] : **FIFOQueue** of $Event$
2  prev[$Proc$] : $Event$ initialized to $\bot$
3
4  **monitor**$_{PSO}$(e):
5    // *Could simulation have $\rightarrow_{hb}-$cycle?*
6    **if** $\exists e' \in B[p(e')][a(e)]$ **with**
           $p(e') \neq p(e) \wedge e' \rightarrow_{hb}^{*}$prev$[p(e)]$:
7      **signal** "SC Violation"
8
9    // *Ensure equivalence to SC.*
10   **while** $\exists p' \neq p(e)$ **with not**
                       $B[p'][a(e)].empty()$:
11     $B[p'][a(e)].removeFirst()$
12     **emit** $\mathbf{st}^c(p', a(e))$
13
14   // *Execute e in PSO simulation.*
15   prev$[p(e)] := e$
16   **if** $e =$**st**$(p, a, v)$:
17     $B[p][a].addLast(e)$
18     **emit** **st**$(p, a, v)$
19   **else if** $e =$**ld**$(p, a)$:
20     **emit** **ld**$(p, a)$
21   **else if** $e =$**atm**$(p, a, f)$:
22     **while not** $B[p][a].empty()$:
23       $B[p][a].removeFirst()$
24       **emit** $\mathbf{st}^c(p, a)$
25     **emit** **atm**$(p, a, f)$
```

Fig. 5. Monitoring algorithm for PSO

But first, Line 6 of **monitor**$_{TSO}(e)$, resp. **monitor**$_{PSO}(e)$, checks if we can create a violation of sequential consistency by executing memory operation e in the TSO (resp. PSO) simulation *before* committing any pending and conflicting stores. That is, suppose e is an operation on address a by process p, and in our simulated TSO (resp. PSO) execution there is a pending store e' to a by process $p' \neq p$. In the TSO (resp. PSO) execution, we can force $e \rightarrow_c e'$ by executing e (and committing e, if it is a store) before committing e'. Further, suppose that e' satisfies the rest of the condition at Line 6. That is, $e' \rightarrow_{hb}^{*}$ **prev**$[p]$—in the trace of the SC execution, event e' *happens before* the event **prev**$[p]$ issued by process p just before e. Then, as proved in Theorem 1, in the trace of the simulated TSO (resp. PSO) execution we will have $e' \rightarrow_{hb}^{*}$ **prev**$[p] \rightarrow_p e \rightarrow_c e'$. This is a cycle in the $\rightarrow_{hb}$-relation, indicating that the simulated TSO (resp. PSO) execution is not sequentially consistent.

In order to track the classic $\rightarrow_{hb}^{*}$ relation on the trace of the SC execution that we are monitoring, we use a well-known vector clock algorithm. The algorithm has a time complexity of $O(N \cdot P)$ on a trace/execution of length N with P processes. A short description of the algorithm can be found in, e.g., [11].

Theorem 1 *Algorithms* $\boldsymbol{monitor}_{TSO}$ *and* $\boldsymbol{monitor}_{PSO}$ *are sound monitoring algorithms for TSO and PSO, respectively. That is, whenever either reports a violation of sequential consistency given an SC execution of a program* P*, then* P *really has a violation of sequential consistency under TSO (resp. PSO).*

Theorem 2 *Algorithms* $\boldsymbol{monitor}_{TSO}$ *and* $\boldsymbol{monitor}_{PSO}$ *are complete for TSO and PSO, respectively. That is, if program* P *has a violation of sequential consistency under TSO (resp. PSO), then there exists some SC execution of* P *on which* $\boldsymbol{monitor}_{TSO}$ *(resp.* $\boldsymbol{monitor}_{PSO}$*) reports an SC violation.*

Theorem 3 *On a sequentially consistent execution of length* N *on* P *processes, monitoring algorithms* $\boldsymbol{monitor}_{TSO}$ *and* $\boldsymbol{monitor}_{PSO}$ *run in time* $O(N \cdot P)$*.*

We sketch the proofs of these results in the Appendix. Complete proofs can be found in our accompanying technical report.

6 Comparison to SOBER

Our work is inspired by SOBER [4], a previous monitoring algorithm that detects program executions under TSO that violate sequential consistency by examining only SC executions. SOBER is derived from the axiomatic characterization of relaxed memory model TSO, while we work from operational definitions of TSO and PSO. There are four key differences between our work and SOBER.

First, we give monitor algorithms for detecting sequential consistency violations under both TSO and the more relaxed PSO memory model, while SOBER detects only violations under TSO.

Second, the run-time complexity of our algorithms is $O(N \cdot P)$, where P is the number of processors and N is the length of the monitored SC execution. This is an improvement over the complexity $O(N \cdot P^2)$ of the SOBER algorithm. The additional factor of $O(P)$ in SOBER is from a vector clock algorithm to maintain the *relaxed happens-before relation*, which axiomatically defines the behaviors legal under TSO. In contrast, when working from our operational definitions for TSO and PSO, there is no need for such additional vector clocks.

Third, the SOBER monitoring algorithm is more sensitive than our $\mathbf{monitor}_{TSO}$. That is, for some programs there exist individual sequentially consistent executions for which SOBER will report the existence of an SC violation while $\mathbf{monitor}_{TSO}$ will not. However, this does not affect the completeness of our monitoring algorithms—for such programs, there will always exist some other SC execution on which $\mathbf{monitor}_{TSO}$ will detect and report a violation of sequential consistency. In our experimental evaluation, this reduced sensitivity does not seem to hinder our ability to find violations of sequential consistency when combining our monitors with preemption-bounded model checking [19].

Fourth, we believe that working with operational definitions for relaxed memory models TSO and PSO is both simpler than working with axiomatic definitions and leads to more natural and intuitive monitoring algorithms. As evidence for this belief, we note that we have discovered [8] a subtle error in the axiomatic definition of TSO given in [4], which leads SOBER to fail to detect some real violations of sequential consistency under TSO. This error has been confirmed [7] both by the authors of [4] and by [18]. We discuss this error, and how to correct it, in our accompanying technical report.

7 Experimental Evaluation

In order to experimentally evaluate our monitor algorithms, we have implemented $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ on top of the THRILLE [16] tool for model checking, testing, and debugging parallel C programs.

In our experiments, we use seven benchmarks. The names and sizes of these benchmarks are given in Columns 1 and 2 of Tables 1 and 2. Five are implementations of concurrent data structures taken from [3]: `msn`, a non-blocking queue, `ms2`, a two-lock queue, `lazylist`, a list-based concurrent set, `harris`, a non-blocking set, and `snark`, a double-ended queue (dequeue). The other two benchmarks are implementations of Dekker's algorithm and Lamport's bakery algorithm for mutual exclusion. Previous research [3, 4] has demonstrated that the benchmarks have sequential consistency violations under relaxed memory models without added memory fences. For each of the benchmarks we have manually constructed a test harness.

In our experimental evaluation, we combine our monitoring algorithms with THRILLE's *preemption-bounded* [19] model checking. That is, we run $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ on all sequentially consistent executions of each

bench	LOC	# inter-leavings	total time	distinct violations under TSO	distinct violations under PSO
dekker	20	79	6.4	3	5
bakery	30	197	42.4	3	4
msn	80	92	7.5	0	3
ms2	80	123	11.0	0	2
harris	160	161	18.8	0	2
lazylist	120	139	14.0	0	4
snark	150	172	15.4	0	4

Table 1. Experimental evaluation of $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ on all interleavings with up to 1 preemption.

bench	LOC	# inter-leavings	total time	distinct violations under TSO	distinct violations under PSO
dekker	20	1714	180.0	9	11
bakery	30	13632	3992.4	3	4
msn	80	2300	196.1	0	3
ms2	80	3322	300.0	0	2
harris	160	5646	661.7	0	2
lazylist	120	4045	428.4	0	4
snark	150	6510	609.9	0	10

Table 2. Experimental evaluation on all interleavings with up to 2 preemptions.

benchmark with a bounded number of preemptive context switches. This verification is not complete—because we do not apply our monitor algorithms to *every* SC execution, we may miss possible violations of SC under TSO or PSO. We evaluate only whether our monitoring algorithms are effective in finding real violations of SC when combined with a systematic but incomplete model checker.

We run two sets of experiments, one with a preemption bound of 1 and the other with a preemption bound of 2. Columns 3 and 4 list the number of parallel interleavings explored and the total time taken with a preemption bound of 1 (Table 1) and a bound of 2 (Table 2). The cost of running our unoptimized monitor implementations on every sequentially consistent execution adds an overhead of roughly 20% to THRILLE's model checking for the data structure benchmarks and about 100% to the mutual exclusion benchmarks.

Rather than report every single parallel interleaving on which one of our monitor algorithms signaled a violation, we group together violations caused by the same pair of operations e and e'. We say that a violation is *caused by* e and e' when $\mathbf{monitor}_{TSO}(e)$ or $\mathbf{monitor}_{PSO}(e)$ is the call on which a violation is signaled, and e' is the conflicting memory access identified in the condition at Line 6. For such a violation, e' *happens before* the event $\mathbf{prev}[p(e)]$ just before e in process $p(e)$, but event e also *happens before* e' because we delay store e' until after e completes in the violating TSO or PSO execution.

Columns 4 and 5 of Tables 1 and 2 list the number of such distinct violations of sequential consistency found under TSO and PSO in our experiments. Note that we find no violations of sequential consistency under TSO for any of the data structure benchmarks. Their use of locks and compare-and-swap operations appear to be sufficient to ensure sequential consistency under TSO. On the other hand, we find violations of sequential consistency for all benchmarks under PSO.

Acknowledgments We would like to thank Krste Asanović, Pallavi Joshi, Chang-Seo Park, and our anonymous reviewers for their valuable comments. This research supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

References

1. Arvind, A., Maessen, J.W.: Memory model = instruction reordering + store atomicity. In: ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture. pp. 29–40. IEEE Computer Society (2006)
2. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: The 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL) (2010)
3. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: ACM SIGPLAN conference on Programming Language Design and Implementation (2007)

4. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV '08: Proceedings of the 20th international conference on Computer Aided Verification (2008)
5. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. Tech. Rep. MSR-TR-2008-12, Microsoft Research (2008)
6. Burckhardt, S., Musuvathi, M.: Memory model safety of programs. In: $(EC)^2$: Workshop on Exploting Concurrency Efficiently and Correctly (2008)
7. Burckhardt, S., Musuvathi, M.: Personal communcation (2010)
8. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency in relaxed memory models. Tech. Rep. UCB/EECS-2010-31, EECS Department, University of California, Berkeley (Mar 2010), http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-31.html
9. Dill, D.L., Park, S., Nowatzyk, A.G.: Formal specification of abstract memory models. In: Symposium on Research on Integrated Systems (1993)
10. Flanagan, C., Freund, S.N.: Adversarial memory for detecting destructive races. In: ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI) (2010)
11. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05). pp. 110–121 (2005)
12. Gibbons, P., Korach, E.: The complexity of sequential consistency. In: Fourth IEEE Symposium on Parallel and Distributed Processing. pp. 317–235 (1992)
13. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or not QB: An efficient execution verification tool for memory orderings. In: In Computer-Aided Verification (CAV), LNCS 3114 (2004)
14. Higham, L., Kawash, J., Verwaal, N.: Weak memory consistency models. part i: Definitions and comparisons. Tech. Rep. 97/603/05, Department of Computer Science, The University of Calgary (1998)
15. Huynh, T.Q., Roychoudhury, A.: Memory model sensitive bytecode verification. FMSD 31(3), 281–305 (2007)
16. Jalbert, N., Sen, K.: A trace simplification technique for effective debugging of concurrent programs. In: The 18th ACM SIGSOFT international symposium on the Foundations of Software Engineering (SIGSOFT 10/FSE-18) (2010)
17. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (1979)
18. Mador-Haim, S., Alur, R., Martin, M.M.: Generating litmus tests for contrasting memory consistency models. In: CAV '10: Proceedings of the 22nd international conference on Computer Aided Verification (Jul 2010)
19. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. ACM (2007)
20. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: ACM symposium on Parallel algorithms and architectures. pp. 34–41. ACM (1995)
21. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
22. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)
23. SPARC International: The SPARC architecture manual (v. 9). Prentice-Hall (1994)

A Soundness and Completeness Proof Sketches

We sketch here proofs of the soundness and completeness of our monitoring algorithms. Complete proofs can be found in our accompanying technical report.

Theorem 1 *Algorithms* $\textbf{monitor}_{TSO}$ *and* $\textbf{monitor}_{PSO}$ *are sound monitoring algorithms for TSO and PSO, respectively.*

Proof (sketch). Let $(\sigma_0, m_0) \xrightarrow{e_1, r_1} \cdots \xrightarrow{e_n, r_n} (\sigma_n, m_n)$ be an SC execution of a program P such that $\textbf{monitor}_{TSO}$ (resp., $\textbf{monitor}_{PSO}$), on this execution, first signals "Sequential Consistency Violation" on event e_n. We prove that P has an execution under TSO (resp., PSO) with a non-sequentially-consistent trace.

Observe that, during its execution, $\textbf{monitor}_{TSO}$ (resp., $\textbf{monitor}_{PSO}$) emits labels from *Event* and τ_{TSO} (resp., τ_{PSO}). We can show:

(1) That the events emitted during the first $n-1$ calls to $\textbf{monitor}_{TSO}$ (resp., $\textbf{monitor}_{PSO}$) form a TSO (resp., PSO) execution of program P. Further, this execution has the same trace as the SC execution.
We can show that the emitted $\textbf{st}^c$ transitions ensure that the loads and atomic operations in the TSO (resp., PSO) execution see the same values as in the SC execution. Thus, program P behaves identically.

(2) That operation e_n can be performed in this TSO (resp., PSO) execution in a way that creates a trace with a cycle in its $\rightarrow_{hb}$-relation.
In the emitted TSO (resp., PSO) execution, the event e' in the condition at Line 6 is still pending. Thus, we can perform e (and commit it, if it is a store) before e' commits, so that $e \rightarrow_c e'$. And $e' \rightarrow_{hb}^* \textbf{prev}[p(e)]$ because the emitted execution has the same trace as the SC execution $e_1, \ldots, e_{n-1}$.
Thus, we create a TSO (resp., PSO) execution with happens-before cycle:

$$e' \ \rightarrow_{hb}^* \ \textbf{prev}[p(e)] \ \rightarrow_p \ e \ \rightarrow_c \ e'$$

Theorem 2 *Algorithms* $\textbf{monitor}_{TSO}$ *and* $\textbf{monitor}_{PSO}$ *are complete monitoring algorithms for TSO and PSO, respectively.*

Proof (sketch). Suppose $(s_0, m_0) \xrightarrow{l_1} \cdots \xrightarrow{l_n} (s_n, m_n)$ is a TSO (resp., PSO) execution of program P with a trace $(E, \rightarrow_p, src, \rightarrow_{st})$ that is not sequentially consistent. Recall that each l_i is either a memory event from *Event* or an internal transition $\textbf{st}^c(p)$ for TSO or $\textbf{st}^c(p, a)$ for PSO.

We can obtain shorter TSO (resp., PSO) executions of P by removing some *Event*-labeled transition from the execution $l_1, \ldots, l_n$, as well as possibly removing corresponding $\textbf{st}^c$ transitions. For example, we can safely remove the last *Event* issued by any process p, even if it is not last in the execution, as long as it does not write a value that is read by a later operation.

We use this freedom to construct a shorter TSO (resp., PSO) execution that is a *minimal* violation of sequential consistency. That is, if any further *Event*'s are removed, the trace of the execution becomes sequentially consistent.

On this minimally-violating execution, consider the *Event*'s that can be safely removed—i.e. their removal leaves a valid TSO (resp., PSO) execution, but this execution has an SC trace. For such a safely-removable e, let $\mathit{last}(e)$ denote the last write to $a(e)$ to become globally visible in the TSO (resp., PSO) execution, not including e itself. In the TSO (resp., PSO) execution $e \to_c \mathit{last}$, but we would have $\mathit{last}(e) \to_{hb}^* e$ in the SC execution in which e is removed and then run at the end. We can show that, for at least one of the these e, no other event comes after $\mathit{last}(e)$ in any SC trace and also forces $\mathit{last}(e)$ to be committed before event e executes. Thus, $\mathbf{monitor}_{TSO/PSO}(e)$ reports a violation on this execution.

B Complexity of Monitoring Algorithms

Lemma 1. *During the execution of* $\boldsymbol{monitor}_{TSO}$ *(respectively,* $\boldsymbol{monitor}_{PSO}$*), for each address* $a \in \mathit{Adr}$*, at any given time at most one process* $p \in \mathit{Proc}$ *will have pending stores to address* a *in its write buffer* $B[p]$ *(resp.,* $B[p][a]$*).*

Proof (by induction). Initially, before any calls to $\mathbf{monitor}_{TSO}$ (resp., $\mathbf{monitor}_{PSO}$), the lemma clearly holds.

Suppose the lemma holds after k calls to $\mathbf{monitor}_{TSO}$ (resp., $\mathbf{monitor}_{PSO}$), and let $\mathbf{monitor}_{TSO}(e)$ or $\mathbf{monitor}_{PSO}(e)$ be the $(k+1)$-st call. If e is not a store, or if $e = \mathbf{st}(p, a, v)$ where no other process $p' \neq p$ has any pending stores to a, then the lemma clearly holds during and after the call.

Suppose instead that $e = \mathbf{st}(p, a, v)$ and process $p' \neq p$ is the only process with pending stores to address a. Then, in $\mathbf{monitor}_{TSO}$ (resp., $\mathbf{monitor}_{PSO}$), the `while`-loop at Lines 10–12 commits all such pending stores by p' before Line 17 adds a pending store to a to a write buffer of process p.

By Lemma 1, in the condition at Line 6 in $\mathbf{monitor}_{TSO}(e)$ and $\mathbf{monitor}_{PSO}(e)$ at most one processor p' can have pending writes to $a(e)$.

We further observe that $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ remain complete if, at Line 6, we check this condition with only the last (i.e. most recent) pending store to a in $B[p']$ or $B[p'][a]$. See the proof of Theorem 2 for details.

Theorem 3 *On a sequentially consistent execution of length* N *on* P *processes, monitoring algorithms* $\boldsymbol{monitor}_{TSO}$ *and* $\boldsymbol{monitor}_{PSO}$ *run in time* $O(N \cdot P)$*.*

Proof. We show that each call $\mathbf{monitor}_{TSO/PSO}(e)$ runs in amortized $O(P)$ time, yielding $O(N \cdot P)$ total time. As mentioned in the previous section, updating the vector clocks to maintain the happens-before relation on the monitored SC execution requires $O(P)$ time per call to $\mathbf{monitor}_{TSO/PSO}(e)$.

For each address $a \in \mathit{Adr}$, we track the single process p which has pending stores to a. Further, for a we maintain a pointer into the **FIFOQueue** of this process to the last (i.e. most recent) pending store to a. We can maintain these two pieces of per-address information in $O(1)$ time per call to $\mathbf{monitor}_{TSO/PSO}(e)$. Using this information, checking the condition at Line 6 requires $O(1)$ time to find the last pending store to $a(e)$ and $O(1)$ time to check $e' \to_{hb}^* e$. The condition at Line 10 can similarly be checked in $O(1)$.

Finally, the total number of iterations of the `while`-loops at Lines 10 and 22, across all N calls to $\mathbf{monitor}_{TSO/PSO}(e)$, cannot exceed $O(N)$ as we buffer no more than N writes.