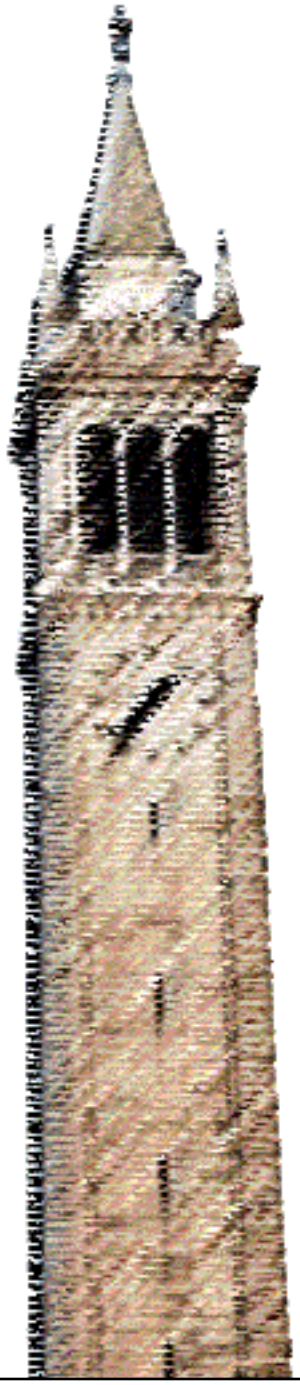
Sound and Complete Monitoring of Sequential Consistency in Relaxed Memory Models



Jacob Burnim
Koushik Sen
Christos Stergiou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-31
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-31.html

March 18, 2010

Sound and Complete Monitoring of Sequential Consistency in Relaxed Memory Models

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

Abstract. We present monitor algorithms that can detect the presence of program executions that violate sequential consistency under the relaxed memory models TSO (Total Store Order) and PSO (Partial Store Order) by examining only the sequentially-consistent executions of a program. Our algorithms are sound and complete—if a program can exhibit a sequential consistency violation under TSO or PSO, the corresponding monitor algorithm can detect it on some sequentially-consistent execution. The monitor algorithms arise naturally from the operational semantics of these relaxed memory models, highlighting an advantage of viewing relaxed memory models operationally rather than axiomatically. We apply our algorithms to several concurrent data structures and synchronization primitives, identifying a number of violations of sequential consistency.

1 Introduction

Programmers writing concurrent software often assume that the underlying memory model is sequentially consistent. That is, that programs will behave as if the executed operations of each thread were interleaved in a single and total sequential order. However, sequential consistency strongly constrains the ordering of memory operations, which can make it difficult to achieve high performance in commodity microprocessors [5, 12]. Thus, to enable increased concurrency and performance, processors often provide a *relaxed* memory model. For example, a processor may delay a store relative to subsequent loads. Unfortunately, working with relaxed memory models often requires subtle and difficult reasoning [5, 12].

Nevertheless, developers of high-performance concurrent programs, such as lock-free data-structures and synchronization libraries, often use regular load and store operations and atomic compare-and-swap-like primitives instead of locks to increase performance. Due to the absence of locks, such implementations have inherent data races and are prone to bugs resulting from the use of relaxed memory models. Bugs due to concurrency and data races are notoriously hard to detect and debug; relaxed memory models make the situation even worse.

Recently, Burckhardt and Musuvathi [3] have argued that programmers, despite using ad-hoc synchronization, expect their program to be sequentially consistent. Therefore, in order to find memory model bugs, it is often sufficient to detect violations of sequential consistency (SC). They proposed an elegant run-time monitoring based approach, called SOBER, to detect SC violations. A key observation made in this work is that for the total store-order (TSO)

memory model if a program execution violates sequential consistency (SC), then there exists a sequentially consistent execution of the program on which run-time monitoring will report a violation of SC. Therefore, if run-time monitoring is performed along with a traditional model checker, which explores all sequentially consistent executions (more specifically, partial-orders) of the program, then all violations of sequential consistency under TSO can be detected. SOBER uses axiomatic semantics of SC and TSO to derive the monitoring algorithm.

In this paper, we propose novel run-time monitoring algorithms for two SPARC [13] memory models: total-store order (TSO) and partial-store order (PSO). TSO allows to delay a store relative to subsequent loads on a processor. The operational model of TSO uses a store buffer for each processor. PSO additionally allows to delay stores relative to other stores to different addresses on the same processor. The operational model of PSO is implemented by using a separate buffer for each address per processor.

A key highlight of our run-time monitoring algorithm is that we use operational semantics [10] of TSO and PSO instead of using their axiomatic semantics. Specifically, we consider a sequentially consistent execution of the program and simulate the execution on the operational model of TSO and PSO by buffering stores as long as they generate the same trace as the SC execution. At the time of simulation of the execution, we also check if the sequential consistency of the execution could be violated by delaying the commit of a pending store in the buffer beyond a subsequent load or store. If such a SC violation is observed, then we report it. We prove the soundness of our algorithms, i.e. if our TSO monitoring algorithm (resp. PSO monitoring algorithm) detects a SC violation, then there exists an execution of the program under TSO (resp. PSO) that violates sequential consistency. We also prove the completeness of our algorithms, i.e. if a program has an execution under TSO (resp. PSO) that violates sequential consistency, then there exists a sequentially consistent execution of the program on which our TSO monitoring algorithm (resp. PSO monitoring algorithm) will detect a SC violation. We show that the complexity of our algorithm is $O(|Proc|.|E|)$, where $|E|$ is the length of the SC trace being monitored and $Proc$ is the set of processors involved in the execution. This complexity is an improvement over the runtime complexity $O(|Proc|^2.|E|)$ of the SOBER algorithm. We have implemented our monitoring algorithms for C programs in a prototype tool and applied it to two mutual exclusion algorithms and several concurrent data structures. Our experiments show that our monitors can detect sequential consistency violations in these benchmarks.

Note that a monitoring algorithm based on axiomatic semantics requires the design of vector clocks for each relaxed memory model in addition to the standard vector clocks to track traditional happens-before relation. In contrast, we do not need to design a vector clock algorithm for each relaxed memory model—we use the operational model directly to find SC violations. As such our algorithm is fundamentally different from SOBER [3]. We also illustrate (in Section 4) that we can find sequential consistency violations in TSO executions that SOBER cannot detect because SOBER uses a definition of the TSO memory model that is more restrictive than the standard one [13].

Other Related Work. There have been a number of efforts to verify and model check concurrent programs on relaxed memory models [5, 12, 8, 14, 2, 9]. Some of these techniques [8, 14, 2] encode a program as well as the axiomatic semantics of the underlying memory model as a constraint system and use a constraint solver to find bugs. Other techniques [5, 12, 9] explicitly explore the state space of a program to find bugs.

Recently, Flanagan and Freund proposed to use an adversarial memory to discover at run-time if a data race in a Java program could be harmful under Java's relaxed memory model [6]. For a read operation involved in a data race, the technique checks if there exists a different return value allowed by the Java Memory Model, that would cause the program to exhibit a bug.

2 Problem Formulation

In this section, we formally define the problem of monitoring sequentially-consistent executions to detect violations of sequential consistency in executions under relaxed memory models.

2.1 Memory Traces

A memory trace (or simply a *trace*) is a record of the memory operations issued by some collection of processors to an underlying memory model. A trace can be thought of as a set of memory operations, called *events*.

Following [3] and [4], we formally define a trace to be a tuple $E = (I, R, W, L, src, c)$, where:

- $I \subseteq l \times Proc \times Adr \times \mathbb{N}$ is the set of *events*, where l is a set of instruction identifiers, $Proc = \{1, \ldots, P\}$ is a set of processor identifiers, Adr is a set of memory addresses, and $\mathbb{N}$ is the set of natural numbers. An element (l, p, a, i) in I denotes that the i^{th} memory operation issued by instruction labeled l by processor p accesses (e.g. reads or writes) memory location a.
- $R \subseteq I$ is the set of all events that read from memory.
- $W \subseteq I$ is the set of all events that write to memory.
- $L \subseteq I$ is the set of all interlocked events[1].
- src is a partial function $R \rightharpoonup W$. We write $src(e) = \perp$ to indicate that src is not defined for some event $e \in R$.
- $c \subseteq W \times W$ is a relation such that: (1) for each address $a \in Adr$, c is a total order on the writes of a, and (2) c does not relate writes to different addresses.

Further, to be a trace E must satisfy the following requirements:

1. $I = R \cup W$ and $L = R \cap W$.
2. If (l, p, a, i) is present in I, then for each $1 \leq j < i$, an event of the form (l', p, a', j) must also be present in I.
3. In a trace, if r is in R and $src(r)$ is defined, then $src(r)$ must be in W.

[1] An interlocked operation [3] can be used to model compare-and-swap and fences.

For each event $e \in I$, we define projection functions l, p, a, and i that return the instruction id, processor id, address, and issue index, respectively, of e.

For a trace, function *src* models the flow of values from store events to load events. That is, $src(e) = e'$ models that load e reads from the memory the value written by store e'.

Similarly, the relation c models the total order in which the stores on any given address $a \in Adr$ become globally visible. Thus, when $(e, e') \in c$ for $e, e' \in W$ with $a(e) = a(e')$, we say that store e *commits* before store e'. Not all relaxed memory models guarantee the existence of such a global, total ordering on the writes to an address. The TSO and PSO models, however, *do* guarante the existence of such an ordering, justifying the use of this formalism and terminology.[2]

We also define a *restriction* of a trace:

Definition 1. *Let $E = (I, R, W, L, src, c)$ be a trace and let $I' \subseteq I$. The* **restriction of E to I'** *is the tuple $E\,|_{I'} = (I', R', W', L', src', c')$, where $R' = R \cap I'$, $W' = W \cap I'$, $L' = L \cap I'$, $c' = c \cap (W' \times W')$ and src' is the restriction of src to the domain R'.*

2.2 Operational Memory Models

In this section, we give operational semantics for three memory models: sequential consistency (SC) and relaxed memory models TSO and PSO.

We describe a memory model *MM* operationally as an automaton which outputs tuples (I, R, W, L, src, c). An automaton for memory model *MM* has labeled transitions $\mathbf{load}_{MM}(l, p, a)$, $\mathbf{store}_{MM}(l, p, a)$, $\mathbf{interlocked}_{MM}(l, p, a)$, $\mathbf{store}^{c}_{MM}(p)$, and $\mathbf{store}^{c}_{MM}(p, a)$, parameterized by instruction identifiers l, processor identifiers p, and addresses a.

Definition 2. *We say that a trace E is* ***a trace under memory model MM*** *when there exists a run of the automaton for MM that outputs E.*

Definition 3. *In particular, a trace E is* ***sequentially consistent*** *if there exists a run of SC that outputs E.*

We think of the transitions $\mathbf{load}_{MM}(l, p, a)$, $\mathbf{store}_{MM}(l, p, a)$, $\mathbf{interlocked}_{MM}(l, p, a)$ as *input* transitions, which are called by processor p to issue a load, store, or interlocked operation. In this view, $\mathbf{store}^{c}_{MM}$ is an internal transition taken by *MM* when it chooses to commit a store.

Note that we have abstracted out the actual values written to and read from memory. The *src* function in the output tuple (I, R, W, L, src, c) captures from which write each read gets its value.

We now formally describe our operational models for sequential consistency (SC), Total Store Order (TSO), and Partial Store Order (PSO), given in Figures 1, 2, and 3.

[2] This property is closely related to *store atomicity* [1]. The TSO and PSO memory models do not technically have store atomicty, however, because loads see the values of earlier and uncommitted stores on the same processesor.

Sequential Consistency. The operational model for SC is given in Figure 1. In order to generate a trace, the SC automaton tracks the last global write event $m[a]$ to each address a, the number of events $i[p]$ issued by each processor p, and the total number *cnt* of issued write events. The automaton's output is the tuple (I, R, W, L, src, c). Function *src* and sets I, R, W, and L are all constructed as the $\mathbf{store}_{SC}$, $\mathbf{load}_{SC}$, and $\mathbf{interlocked}_{SC}$ transitions are executed. Instead of building relation c directly, the SC automaton assigns increasing *commit indices* $C(e)$ to write events e, and the output relation c is defined in terms of commit indices.

Total Store Order. The operational model for TSO is given in Figure 2. The model is same as the one described and proved equivalent to axiomatic semantics of TSO in [3]. The Total Store Order (TSO) memory model [13] allows stores to be reordered past later loads, but maintains a total order over stores. To maintain intra-thread consistency, when a store is reordered past a load of the same address, the load still sees the stored value.

The TSO automaton tracks the last *committed* write event $m[a]$ to each address a. We introduce per-processor store buffers $B[p]$, first-in first-out queues that hold write events that have been issued but not yet committed. Writes e are added to $B[p]$ in $\mathbf{store}_{TSO}$ and removed, committed to $m[a(e)]$, and assigned commit indices $C(e)$ in $\mathbf{store}^{c}_{TSO}$ and $\mathbf{interlocked}_{TSO}$.

m : **array**$[Adr]$ **of** Evt **initialized to** $\perp$
i : **array**$[Proc]$ **of** $\mathbb{N}$ **initialized to** 0
cnt: $\mathbb{N}$ **initialized to** 0
I, R, W, L : **sets of** Evt **initialized to** $\emptyset$
src, C : **maps initialized to** $empty$ **maps**

store$_{SC}(l, p, a)$:
  $i[p] = i[p] + 1$; $cnt = cnt + 1$; $m[a] = (l, p, a, i[p])$
  add event $m[a]$ **to** I **and** W ; **add** $m[a] \mapsto cnt$ **to** C

load$_{SC}(l, p, a)$:
  $i[p] = i[p] + 1$
  add event $(l, p, a, i[p])$ **to** I **and** R ; **add** $(l, p, a, i[p]) \mapsto m[a]$ **to** src

interlocked$_{SC}(l, p, a)$:
  $i[p] = i[p] + 1$; $cnt = cnt + 1$; $m[a] = (l, p, a, i[p])$
  add event $m[a]$ **to** I , R , W , **and** L
  add $(l, p, a, i[p]) \mapsto m[a]$ **to** src ; **add** $m[a] \mapsto cnt$ **to** C

output : $E = (I, R, W, L, src, c)$ **such that** $c(w_1, w_2)$ **if**
  $C(w_1) \leq C(w_2)$ **and** $a(w_1) = a(w_2)$.

Fig. 1: Operational Model of SC: (I, R, W, L, src, c) is the memory trace

```
m   : array[Adr] of Evt initialized to ⊥
B   : array[Proc] of FIFOQueue of Evt
i   : array[Proc] of ℕ initialized to 0
cnt: ℕ initialized to 0
I, R, W, L : sets of Evt initialized to ∅
src, C : maps initialized to empty maps

store_TSO(l, p, a) :
    i[p] = i[p] + 1 ; B[p].addLast((l, p, a, i[p]))
    add event (l, p, a, i[p]) to I and W

store^c_TSO(p) :
    if not B[p].empty()
        m[a] = B[p].removeFirst() ; cnt = cnt + 1 ; add m[a] ↦ cnt to C

load_TSO(l, p, a) :
    i[p] = i[p] + 1 ; cnt = cnt + 1
    add event (l, p, a, i[p]) to I and R ; add (l, p, a, i[p]) ↦ cnt to C
    if B[p].contains((∗, ∗, a, ∗)
       e = last event (∗, ∗, a, ∗) of B[p] ; add (l, p, a, i[p]) ↦ e to src
    else
       add (l, p, a, i[p]) ↦ m[a] to src

interlocked_TSO(l, p, a) :
    while not B[p].empty()
        store^c_TSO(p)
    i[p] = i[p] + 1 ; cnt = cnt + 1 ; m[a] = (l, p, a, i[p])
    add event m[a] to I , R , W , and L
    add (l, p, a, i[p]) ↦ m[a] to src ; add m[a] ↦ cnt to C

output : E = (I, R, W, L, src, c) such that c(w1, w2) if
         C(w1) ≤ C(w2) and a(w1) = a(w2) .
```

Fig. 2: Operational Model of TSO: (I, R, W, L, src, c) is the memory trace

Note that in $\mathbf{load}_{TSO}$, the *src* of a read event by processor p can be a write event in p's write buffer $B[p]$. This captures that reads can see values of pending writes from the same process.

Note also that, to facilitate later reasoning, we increment *cnt* on reads as well as writes and also assign commit indices $C(e)$ to reads.

We think of $\mathbf{store}^c_{TSO}$ as an *internal* transition. A program running under TSO issues writes by calling $\mathbf{store}_{TSO}$, but the writes are not committed until the memory model non-deterministically chooses to execute corre-

```
m   : array[Adr] of Evt initialized to ⊥
B_a : array[Proc][Adr] of FIFOQueue of Evt
i   : array[Proc] of ℕ initialized to 0
cnt: ℕ initialized to 0
I, R, W, L : sets of Evt initialized to ∅
src, C : maps initialized to empty maps

store_PSO(l, p, a) :
    i[p] = i[p] + 1 ; B_a[p][a].addLast((l, p, a, i[p]))
    add event (l, p, a, i[p]) to I and W

store^c_PSO(p, a) :
    if not B_a[p][a].empty()
      m[a] = B_a[p][a].removeFirst() ; cnt = cnt + 1 ; add m[a] ↦ cnt to C

load_PSO(l, p, a) :
    i[p] = i[p] + 1 ; cnt = cnt + 1
    add event (l, p, a, i[p]) to I and R ; add (l, p, a, i[p]) ↦ cnt to C
    if not B_a[p][a].empty()
        e = B_a[p][a].getLast() ; add (l, p, a, i[p]) ↦ e to src
    else
        add (l, p, a, i[p]) ↦ m[a] to src

interlocked_PSO(l, p, a) :
    while not B_a[p][a].empty()
        store^c_PSO(p, a)
    i[p] = i[p] + 1 ; cnt = cnt + 1 ; m[a] = (l, p, a, i[p])
    add event m[a] to I , R , W , and L
    add (l, p, a, i[p]) ↦ m[a] to src ; add m[a] ↦ cnt to C

output : E = (I, R, W, L, src, c) such that c(w_1, w_2) if
         C(w_1) ≤ C(w_2) and a(w_1) = a(w_2) .
```

Fig. 3: Operational Model of PSO: (I, R, W, L, src, c) is the memory trace

sponding $\mathbf{store}^c_{TSO}$ transitions. (Or the program forces a commit with an $\mathbf{interlocked}_{TSO}$.)

Partial Store Order. The Partial Store Order (PSO) memory model [13] is similar to TSO, but further allows stores to be reordered past later stores to different addresses.

The PSO automaton [10] is very similar to TSO, but pending writes are stored in per-processor, per-address buffers $B_a[p][a]$. Transition $\mathbf{store}^c_{PSO}(p, a)$ commits the oldest buffered write by p to address a, leaving pending stores to other addresses untouched.

Note that $\mathbf{interlocked}_{PSO}(l, p, a)$ only forces the commit of pending stores to the same address a.

2.3 Operational Program Model

A memory trace of a program P under a memory model MM is produced by the combined execution of the program and of the underlying memory model.

We can think of a program P as consisting of N processes, each with its own purely local state and each running on a distinct processor. In each step of a program execution, one of P's processes issues a single memory operation–e.g. a write or read of an address—to the underlying memory model automaton by calling, $\mathbf{load}_{MM}$, $\mathbf{store}_{MM}$, or $\mathbf{interlocked}_{MM}$. The memory model indicates from which write each read e gets its value via $src(e)$. Then, the local state of the process is updated. The new state must be a deterministic function solely of the previous local state (including, in particular, the process's program counter) and of the value returned by an issued read.

There are two sources of non-determinism in a program execution in this model: (1) The thread schedule of the program. That is, at each step in the execution, which process executes a transition. (2) The internal non-determinism of the memory model. In particular, when the memory model chooses to *commit* issued writes by executing internal $\mathbf{store}^{c}_{MM}$ transitions.

Rather than laboriously formalize this model in order to define when a memory trace is a trace of program P, we propose the following two axioms:

Axiom 1 *Let $E = (I, R, W, L, src, c)$ be a trace of a program P. For $I' \subseteq I$, if the restriction $E\,|_{I'}$ is a trace, then it is a trace of program P.*

Axiom 1 captures the fact that, under any reasonable model of program execution, if a program P can produce a trace E, then a shorter run of P can produce a prefix E' of E. Recall that if $E\,|_{I'}$ is trace, then each processor p issues in E' some prefix of the memory operations it issues in E' and every read e in $E\,|_{I'}$ reads the same store $src(e)$ as in E.

Axiom 2 *Let $E = (I, R, W, L, src, c)$ and $E' = (I, R, W, L, src', c')$ be two traces. If E is a trace of program P and src is equal to src' except for at most one event $e \in R$ where e is the last event issued by some processor, then E' is a trace of the program P.*

When else can we infer, given a trace E of program P, that a second trace E' is also a trace of program P—i.e. also respects the control-flow and logic of P? If a read e is the last memory operation issued by some process in P, then the value returned to e—i.e. $src(e)$—has no impact on the continued execution of P. Thus, Axiom 2 states that if E is a trace of program P, so too is any trace that is identical except for the src function on such a final read.

2.4 The Happens-Before Relation

In reasoning about traces, we will use three relations to capture different kinds of ordering between memory events.

Definition 4. *Let $E = (I, R, W, L, src, c)$ be a trace. Two events $e, e' \in I$ are related by the* **program-order relation***, denoted $e \rightarrow_p e'$ in E, iff $p(e) = p(e')$ and $i(e) < i(e')$.*

That is, $e \rightarrow_p e'$ in a trace when e and e' are events from the same process and e is issued before e'.

Definition 5. *Let $E = (I, R, W, L, src, c)$ be a trace. Two events $e, e' \in I$ are related by the* **conflict-order relation***, denoted $e \rightarrow_c e'$, iff $a(e) = a(e')$ and one of the following holds:*

- *$e \in W$, $e' \in R$, and $e = src(e')$,*
- *$e \in W$, $e' \in W$ and $(e, e') \in c$,*
- *$e \in R$, $e' \in W$, and either $src(e)$ is undefined or $(src(e), e') \in c$.*

Definition 6. *For a trace E, the* happens-before relation *is defined as the union o the program-order and conflict-order relations for E, i.e. $\rightarrow_{hb} = (\rightarrow_p \cup \rightarrow_c)$. We refer to the transitive closure of the happens-before relation as $\rightarrow^*_{hb}$.*

Recall that in Section 2.2, we defined a trace E to be sequentially consistent iff it can be produced by the operational model SC. We can use the happens-before relation to give an alternate characterization of sequential consistency. In particular, a trace E is sequentially consistent iff the $\rightarrow^*_{hb}$ relation is acyclic on the events of E.

Proposition 1. *Let E be a memory trace. Trace E is sequentially consistent iff relation $\rightarrow^*_{hb}$ is acyclic on the events of E.*

We omit a proof of this proposition—the equivalence of these two characterizations of sequential consistency is widely known. For example, the technical report associated with [3] proves a version of this result.

3 Monitoring Algorithm

The monitor algorithms take as input a sequentially consistent trace $E' = (I', R', W', L', src', c')$ together with a total order of the events I'. The total order is a linearization of the $\rightarrow_{hb}$ relation of the trace and denotes the actual sequence of events generated by an execution on a sequentially consistent memory model. We represent the total order as a sequence $e_1, \ldots, e_n$.

Fig 4 describes the monitoring algorithm for TSO and Fig 5 describes the monitoring algorithm for PSO. In these algorithms we make calls to the transitions of a TSO/PSO automaton, respectively, and use the buffers used by the transitions. The algorithms process the events in the sequence $e_1, \ldots, e_n$ one-by-one and call the TSO/PSO transitions so that the trace generated by the TSO/PSO automaton is E'. However, each automaton delays the commit of a store as long as it does not deviate from the generation of the trace E'. These algorithms also check if a pending store in a buffer could be committed after an event that happens after the store. If such a pending store exists, then a sequential consistency violation is reported.

```
1  monitor_TSO(e_1, ..., e_n):
2   for i = 1 to n:
3     if ∃e such that e ∈ B[p(e)] and a(e_i) = a(e):
4       e = last element (∗, ∗, a(e_i), ∗) added to B[p(e)]
5       if ∃prev such that p(prev) = p(e_i) and i(prev) + 1 = i(e_i):
6           if p(e) ≠ p(e_i) and e →*_hb prev:
7               print ''Sequential Consistency Violated''
8       if p(e) ≠ p(e_i):
9           while B[p(e)].first() ≠ e:
10              store^c_TSO(p(e))
11          store^c_TSO(p(e))
12    if e_i ∈ L':
13      interlocked_TSO(l(e_i), p(e_i), a(e_i))
14    else if e_i ∈ R':
15      load_TSO(l(e_i), p(e_i), a(e_i))
16    else if e_i ∈ W':
17      store_TSO(l(e_i), p(e_i), a(e_i))
```

Fig. 4: Monitoring algorithm for TSO

```
1  monitor_PSO(e_1, ..., e_n):
2   for i = 1 to n:
3     if ∃e such that e ∈ B[p(e)][a(e_i)]:
4       if ∃prev such that p(prev) = p(e_i) and i(prev) + 1 = i(e_i):
5           if p(e) ≠ p(e_i) and e →*_hb prev:
6               print ''Sequential Consistency Violated''
7       if p(e) ≠ p(e_i) or e_i ∈ W':
8           while not B[p(e)][a(e)].empty():
9               store^c_PSO(p(e), a(e))
10    if e_i ∈ L':
11      interlocked_PSO(l(e_i), p(e_i), a(e_i))
12    else if e_i ∈ R':
13      load_PSO(l(e_i), p(e_i), a(e_i))
14    else if e_i ∈ W':
15      store_PSO(l(e_i), p(e_i), a(e_i))
```

Fig. 5: Monitoring algorithm for PSO

In order to track the classic $\rightarrow^*_{hb}$ relation we use a well-known vector clock algorithm. The algorithm has a time complexity of $O(|Proc|.|E|)$. A short description of the algorithm can be found in [7] among other papers.

In the following discussion let us fix a program P. Let $\mathcal{T}_{MM}$ be the set of all traces generated by all feasible execution of the program P under the memory model MM, where $MM \in \{SC, TSO, PSO\}$. We next state the correctness results of our monitoring algorithms. The proof of these results can be found in the appendix.

Lemma 1 *During the execution of* $\boldsymbol{monitor}_{TSO}$*, it will never be the case there exists some address* $a \in Adr$ *and two processes* $p' \neq p''$ *with both* $B[p']$ *and* $B[p'']$ *containing a buffered store to* a*.*

Lemma 2 *During the execution of* $\textbf{\textit{monitor}}_{PSO}$*, for each address* $a \in Adr$*, only one process* p *will have at most one buffered store* $B_a[p][a]$ *and for all* $p' \neq p$*,* $B_a[p'][a]$ *will be empty.*

Theorem 3 *(Soundness of PSO Monitoring) If PSO monitor reports a sequential consistency violation", then there exists a trace* $E \in \mathcal{T}_{PSO} \setminus \mathcal{T}_{SC}$ *of the program.*

Theorem 4 *(Soundness of TSO Monitoring) If TSO monitor reports a sequential consistency violation, then there exists a trace* $E \in \mathcal{T}_{TSO} \setminus \mathcal{T}_{SC}$ *of the program.*

Theorem 5 *(Completeness of PSO Monitoring) If there exists a trace* $E \in \mathcal{T}_{PSO} \setminus \mathcal{T}_{SC}$ *of program* P*, then there exists a sequentially consistent trace* E' *of* P *such that PSO monitoring on* E' *reports a violation of sequential consistency.*

Theorem 6 *(Completeness of TSO Monitoring) If there exists a trace* $E \in \mathcal{T}_{TSO} \setminus \mathcal{T}_{SC}$ *of program* P*, then there exists a sequentially consistent trace* E' *of* P *such that TSO monitoring on* E' *reports a violation of sequential consistency.*

Theorem 7 *(Complexity of TSO/PSO Monitoring) Time complexity of both TSO/PSO monitoring is* $O(|Proc|.|E|)$*, where* $|E|$ *is the length of the trace and* $|Proc|$ *is the number of processes.*

4 Comparison to SOBER

Our work is inspired by SOBER [3], a previous monitoring algorithm that detects program executions under TSO that violate sequential consistency by examining only sequentially consistent 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, SOBER uses a definition of the TSO memory model that is more restrictive than the TSO memory model described in the SPARC manual [13]. Consider the simple program given in Figure 6 and the TSO run of this program (with `thread1` on processor p_1 and `thread2` on processor p_2) and resulting trace

```
        Initially: a = b = 0.
thread1                    thread2
 1:  a = 1;                 4:  b = 1;
 2:  if (a == 1)            5:  a = 2;
 3:     print b;            6:  interlock(c);
                            7:  print a;
```

Fig. 6: Example program.

Consider the TSO run:

$$\mathbf{st}_{TSO}(l_1,p_1,a),\ \mathbf{ld}_{TSO}(l_2,p_1,a),\ \mathbf{ld}_{TSO}(l_3,p_1,b),\ \longrightarrow$$
$$\longrightarrow\ \mathbf{st}_{TSO}(l_4,p_2,b),\ \mathbf{st}^c_{TSO}(p_2),\ \mathbf{st}_{TSO}(l_5,p_2,a),\ \mathbf{st}^c_{TSO}(p_2),\ \mathbf{st}^c_{TSO}(p_1)$$

which produces the TSO trace (I, R, W, L, src, c):

$$\begin{aligned} I &= R \cup W \\ R &= \{(l_2,p_1,a,2),\ (l_3,p_1,b,3)\} \\ W &= \{(l_1,p_1,a,1),\ (l_4,p_2,b,1),\ (l_5,p_2,a,2)\} \\ src &= (l_2,p_1,a,2) \mapsto (l_1,p_1,a,1),\ \ (l_3,p_1,b,3) \mapsto \bot \\ c &= \{((l_5,p_2,a,2),\ (l_1,p_1,a,1))\} \cup \{(w,w) :\ w \in W\} \end{aligned}$$

Fig. 7: Example.

given in Figure 7. In this run, the store (l_1, p_1, a) by processor p_1 to address a is buffered until after the later store (l_5, p_2, a) by processor p_2 to address a commits. This trace is not sequentially consistent: although the load (l_3, p_1, b) of b by process p_1 does not see the value written to b by process p_2 in store (l_4, p_2, b), the earlier store (l_1, p_1, a) is not overwritten by the later one (l_5, p_2, a). That is, we have a happens-before cycle: $(l_1, p_1, a) \rightarrow_p (l_3, p_1, b) \rightarrow_c (l_4, p_2, b) \rightarrow_p (l_5, p_2, a) \rightarrow_c (l_1, p_1, a)$.

But, under the axiomatic model of TSO used in SOBER, this trace is not possible. Technically, the TSO model used in SOBER forbids this trace because of a cycle in a subset of $\rightarrow_{hb}$ called the *relaxed happens-before relation*: $(l_2, p_1, a) \rightarrow_p (l_3, p_1, b) \rightarrow_c (l_4, p_2, b) \rightarrow_p (l_5, p_2, a) \rightarrow_c (l_2, p_1, a)$.

There exists no sequentially consistent trace of the program in Figure 6 on which SOBER will report a violation. On the other hand, our monitor algorithm will report a violation given the following sequentially consistent trace from the following SC run:

$$\mathbf{st}_{SC}(l_1,p_1,a),\ \mathbf{ld}_{SC}(l_2,p_1,a),\ \mathbf{ld}_{SC}(l_3,p_1,b),\ \mathbf{st}_{SC}(l_4,p_2,b),\ \mathbf{st}_{SC}(l_5,p_2,a)$$

Third, the runtime complexity of our algorithms is $O(|Proc|.|E|)$, where $Proc$ is the set of processors and $|E|$ is the length of the trace. This is an improvement over the complexity $O(|Proc|^2.|E|)$ of the SOBER algorithm. Note that the factor $|Proc|^2.|E|$ comes from the vector clock algorithm that maintains the relaxed happens-before relation for TSO in SOBER.

Fourth, the SOBER monitoring algorithm is more sensitive than our $\mathbf{monitor}_{TSO}$. That is, there exist sequentially consistent traces E for which SOBER will report the existence of a violation while $\mathbf{monitor}_{TSO}$ will not. Note that this does not mean that $\mathbf{monitor}_{TSO}$ will not find the violation—the violation will be detected by $\mathbf{monitor}_{TSO}$ on some other SC execution (follows from the completeness theorem.) Consider the sequentially consistent run and trace shown in Figure 8. SOBER correctly infers from this trace that there exists a TSO execution in which the store (l_1, p_1, a) to address a in processor p_1 is buffered until after the load (l_5, p_3, a) of a in processor p_3, which is a violation

Consider the sequentially-consistent run:

$$\mathbf{st}_{SC}(l_1, p_1, a),\ \ \mathbf{ld}_{SC}(l_2, p_1, b),\ \ \mathbf{st}_{SC}(l_3, p_2, a),\ \ \mathbf{st}_{SC}(l_4, p_3, b),\ \ \mathbf{ld}_{SC}(l_5, p_3, a)$$

which produces the sequentially-consistent trace (I, R, W, L, src, c):

$$\begin{aligned} I &= R \cup W \\ R &= \{(l_2, p_1, b, 2),\ (l_5, p_3, a, 2)\} \\ W &= \{(l_1, p_1, a, 1),\ (l_3, p_2, a, 1),\ (l_4, p_3, b, 1)\} \\ src &= (l_2, p_1, b, 2) \mapsto \bot,\ \ (l_5, p_3, a, 2) \mapsto (l_3, p_2, a, 1) \\ c &= \{((l_1, p_1, a, 1),\ (l_3, p_2, a, 1))\} \cup \{(w, w) :\ w \in W\} \end{aligned}$$

Fig. 8: Example (and not related to program in Figure 6). This example is used in description of the fourth difference with SOBER.

of sequential consistency. But $\mathbf{monitor}_{TSO}$ will *not* report a violation on this trace. (Because our monitor algorithm will commit processor p_1's store (l_1, p_1, a) to address a when it later encounters processor p_2's store (l_3, p_2, a) to the same address.) Note that this example says nothing about the completeness of our monitor algorithm. We will discover this TSO execution and report a violation on a different SC execution—one in which the store (l_3, p_2, a) in processor p_2 does not occur until after the load (l_5, p_3, a) in processor p_3.

5 Experiments

We have implemented $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ for C programs in a prototype tool. The implementation instruments code to capture load, store, and compare-and-swap operations.

We evaluated our monitor algorithms on seven benchmarks. Five of those are implementations of concurrent data structures taken from [2]: `msn`, a non-blocking queue, `ms2`, a two-lock queue, `lazylist`, a list-based concurrent set, `harris`, a non-blocking set, and `snark`. The other two benchmarks are implementations of the Dekker and Lamport's baker mutual exclusion algorithms. Previous research [2, 3] shows that the benchmarks have sequential consistency violations under relaxed memory models. For each of the benchmarks we have

		average	TSO			PSO		
Benchmark	LOC	trace length	# of traces with violations	# of distinct violations	average runtime	# of traces with violations	# of distinct violations	average runtime
dekker	23	140	973	10	0.01	977	15	0.01
bakery	31	515	992	3	0.04	1000	4	0.05
msn	83	337	-	-	0.01	943	3	0.02
ms2	78	193	-	-	0.01	73	2	0.01
harris	155	1332	-	-	0.06	976	2	0.07
lazylist	121	443	-	-	0.02	18	2	0.02
snark	150	376	-	-	0.01	951	10	0.02

Table 1: Results of $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ experimental evaluation with 1000 random executions per benchmark; all runtimes are in seconds.

either reused test harnesses from [2] or have manually constructed a test harness involving at most 3 threads.

In the absence of a CHESS [11] like model checker for C, we relied on random schedule based execution of the test harnesses to generate 1000 sequentially consistent traces per benchmark. The results of running our algorithms on these traces are shown in Table 1. Column 2 lists the lines of code in each benchmark not including the test harness and column 3 reports the average number of events in the generated traces. Columns 4 and 7 list the number of traces in which $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ respectively detected a sequential consistency violation. For each violation we record the set of program instructions that generated events $e, prev, e_i$, as they are defined in lines 4,5 of $\mathbf{monitor}_{TSO}$ and lines 3,4 of $\mathbf{monitor}_{PSO}$, and in columns 6 and 9 we report the number of violations that were detected and involved a distinct set of those instructions. Columns 7 and 10 report the average runtime of the monitor algorithms for TSO and PSO in seconds. The experimental results show that monitoring is effective in efficiently discovering sequential consistency violations on a set of randomly generated traces. For full experimental evaluation, we plan to combine our work with a PThread model checker, such as Inspect [15], in future work.

References

1. A. Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 29–40. IEEE Computer Society, 2006.
2. S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, 2007.
3. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, 2008.
4. S. Burckhardt and M. Musuvathi. Memory model safety of programs. In *$(EC)^2$: Workshop on Exploting Concurrency Efficiently and Correctly*, 2008.
5. D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, 1993.
6. C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, 2010.
7. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121, 2005.
8. G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *In Computer-Aided Verification (CAV), LNCS 3114*, 2004.
9. T. Q. Huynh and A. Roychoudhury. Memory model sensitive bytecode verification. *FMSD*, 31(3):281–305, 2007.
10. S. Mador-Haim, R. Alur, and M. M. K. Martin. Specifying relaxed memory models for state exploration tools. In *$(EC)^2$: Workshop on Exploting Concurrency Efficiently and Correctly*, 2009.

11. M. Musuvathi and S. Qadeer. 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.
12. S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *ACM symposium on Parallel algorithms and architectures*, pages 34–41. ACM, 1995.
13. SPARC International. *The SPARC architecture manual (v. 9)*. Prentice-Hall, 1994.
14. Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. *Parallel and Distributed Processing Symposium, International*, 1:31b, 2004.
15. Y. Yang, X. X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical Report UUCS-08-004, School of Computing, University of Utah, 2008.

Appendix

Lemma 1 *During the execution of* $\textbf{\textit{monitor}}_{TSO}$*, it will never be the case there exists some address* $a \in Adr$ *and two processes* $p' \neq p''$ *with both* $B[p']$ *and* $B[p'']$ *containing a buffered store to* a*.*

Proof. Suppose there are two events $e' = (l', p', a, i')$ and $e'' = (l'', p'', a, i'')$, in store buffers $B[p']$ and $B[p'']$ respectively, at some point in the execution of $\textbf{monitor}_{TSO}$. Further suppose that the $\textbf{store}_{TSO}$ call that buffered event e' was executed before the call that buffered event e''. Let j be the iteration of the monitor algorithm in which the $\textbf{store}_{TSO}$ that buffered e'' was called. Since e' must have been in $B[p']$ at the start of this iteration, the conditions at Lines 3 and 7 must have evaluated to true with $e = e'$, and thus Lines 8, 9, and 10 were executed. But then Line 10 must have removed e' from $B[p']$, contradicting our initial assumption.

Lemma 2 *During the execution of* $\textbf{\textit{monitor}}_{PSO}$*, for each address* $a \in Adr$*, only one process* p *will have at most one buffered store* $B_a[p][a]$ *and for all* $p' \neq p$*,* $B_a[p'][a]$ *will be empty.*

Proof. Suppose there are two events $e' = (l', p, a, i')$ and $e'' = (l'', p, a, i'')$, in store buffer $B_a[p][a]$ at some point in the execution of $\textbf{monitor}_{PSO}$. Further suppose that the $\textbf{store}_{PSO}$ call that buffered event e' was executed before the call that buffered event e''. Let j be the iteration of the monitor algorithm in which the $\textbf{store}_{PSO}$ that buffered e'' was called. Since e' must have been in $B_a[p][a]$ at the start of this iteration, the conditions at Lines 3 and 7 (first predicate at line 7) must have evaluated to true with $e = e'$, and thus Lines 8 and 9 were executed. But then Line 9 must have removed e' from $B_a[p][a]$, contradicting our initial assumption.

Suppose there are two events $e' = (l', p, a, i')$ and $e'' = (l'', p', a, i'')$, in store buffers $B_a[p][a]$ and $B_a[p'][a]$, respectively, at some point in the execution of $\textbf{monitor}_{PSO}$. Further suppose that the $\textbf{store}_{PSO}$ call that buffered event e' was executed before the call that buffered event e''. Let j be the iteration of the monitor algorithm in which the $\textbf{store}_{PSO}$ that buffered e'' was called. Since e' must have been in $B_a[p][a]$ at the start of this iteration, the conditions at Lines 3 and 7 (second predicate at line 7) must have evaluated to true with $e = e'$, and thus Lines 8 and 9 were executed. But then Line 9 must have removed e' from $B_a[p][a]$, contradicting our initial assumption.

Theorem 3 *(Soundness of PSO Monitoring) If PSO monitor reports a sequential consistency violation", then there exists a trace* $E \in \mathcal{T}_{PSO} \setminus \mathcal{T}_{SC}$ *of the program.*

Proof. Let $E' = (I', R', W', L', src', c')$ be an SC trace of some program P and $e_1, \ldots, e_n$ be a linearization of trace E' that respects the $\rightarrow^*_{hb}$ order defined by E'. Suppose $\textbf{monitor}_{PSO}$ reports a violation of SC on this linearization in iteration

```
1   if e_i ∈ L'
2       interlocked_PSO(l(e_i), p(e_i), a(e_i))
3   else if e_i ∈ R'
4       load_PSO(l(e_i), p(e_i), a(e_i))
5   else if e_i ∈ W'
6       store_PSO(l(e_i), p(e_i), a(e_i))
7       while B_a[p(e_i)][a(e_i)].empty()
8           store^c_PSO(p(e_i), a(e_i))
9   foreach p ∈ Proc and a ∈ Adr
10      while B_a[p][a].empty()
11          store^c_PSO(p, a)
```

Fig. 9: Extra statements for PSO monitoring

j at line number 6. After printing *Sequential Consistency Violated* at line 6, let $\mathbf{monitor}_{PSO}$ execute the following extra statements shown in Figure 9
During this monitoring, $\mathbf{monitor}_{PSO}$ will call a sequence of transitions $\mathbf{store}_{PSO}$, $\mathbf{store}^c_{PSO}$, $\mathbf{load}_{PSO}$, and $\mathbf{interlocked}_{PSO}$. These calls will generate a trace $E = (I, R, W, L, src, c)$. We prove that (1) E is a PSO trace of P and (2) E has a $\rightarrow^*_{hb}$ cycle.

[(1) Proof of E is a PSO trace of P].

Since we call $\mathbf{interlocked}_{PSO}(l(e_i), p(e_i), a(e_i))$, $\mathbf{store}_{PSO}(l(e_i), p(e_i), a(e_i))$, $\mathbf{load}_{PSO}(l(e_i), p(e_i), a(e_i))$ for each e_i in the sequence $e_1, e_2, \ldots, e_j$ if $e_i \in L'$, $e_i \in W'$, and $e_i \in R'$, respectively (see lines 10–15 in Figure 5 and lines 1–8 in Figure 9), I will be the set $\{e_1, e_2, \ldots, e_j\}$, $R = R' \cap I$, $W = W' \cap I$, and $L = L' \cap I$. However, src may not be the function src' restricted to the domain R. Let src'' be the function src' restricted to the domain R. Consider the following two cases:

[Case 1] Let $e_{i1} = src'(e_{i2})$ for some $e_{i2} \in R$ such that $e_{i2} \neq e_j$. Since E' is a SC trace of P and $e_1, \ldots, e_n$ is a linearization of the $\rightarrow^*_{hb}$ relation on E', we have $i1 < i2$. Then we can prove that $src'(e_{i2}) = src(e_{i2})$ as follows.

- If $p(e_{i1}) = p(e_{i2})$ and there is no $e_{i3} \in R'$ such that $i1 < i3 < i2$, $a(e_{i3}) = a(e_{i2})$, and $p(e_{i3}) \neq p(e_{i2})$, then the condition at line 7 is not true. Therefore, e_{i1} will be in $B_a[p(e_{i1})][a(e_{i1})]$ and $src(e_{i2})$ will be e_{i1}, i.e. $src'(e_{i2}) = src(e_{i2})$.
- If $p(e_{i1}) = p(e_{i2})$ and there is a $e_{i3} \in R'$ such that $i1 < i3 < i2$, $i3$ is minimal, $a(e_{i3}) = a(e_{i2})$, and $p(e_{i3}) \neq p(e_{i2})$, then the condition at line 3 is not true because the $i3^{th}$ iteration of $\mathbf{monitor}_{PSO}$ has removed e_{i1} from $B_a[p(e_{i1})][a(e_{i1})]$. Therefore, e_{i1} will be in $m[a(e_{i1})]$ and $src(e_{i2})$ will be e_{i1}, i.e. $src'(e_{i2}) = src(e_{i2})$.
- If $p(e_{i1}) \neq p(e_{i2})$, then e_{i1} will be in $m[a(e_{i1})]$ and $src(e_{i2})$ will be e_{i1}, i.e. $src'(e_{i2}) = src(e_{i2})$.
- There is no $e_{i3} \in W'$ such that $i1 < i3 < i2$ and $a(e_{i3}) = a(e_{i2})$.

[Case 2] Let $e_{i1} = src'(e_j)$. Since the j^{th} iteration reports SC violation, conditions at lines 3, 4, and 5 hold. Therefore, e_{i1} is in $B_a[p(e_{i1})][a(e_{i1})]$ and

$p(e_j) \neq p(e_{i1})$; hence, $\mathbf{load}_{PSO}$ at line 4 in Figure 10 will not read value stored by e_{i1}, i.e. $src(e_j) \neq e_{i1}$.

Therefore, we showed that src is equal to src'' (recall that src'' is the function src' restricted to the domain R) except for at most one event e_j. Note that e_j is the last event of the process $p(e_j)$ in the trace E. Therefore, by Axiom 1, since E' is a trace of the program P, (I, R, W, L, src'', c'') for some c'' is a trace of the program P. Further by Axiom 2, the trace $E = (I, R, W, L, src, c)$ is a trace of the program P. Since E is generated from the operational model of PSO, E is a PSO trace. Therefore, E is a PSO trace of the program P.

[(2) Proof of E has a $\rightarrow_{hb}^*$ cycle].

Let $I_j = I \setminus \{e_j\}$ and $E\ |_{I_j}$ be $E_j = (I_j, R_j, W_j, L_j, src_j, c_j)$. We want to show that if for any $e, e' \in I_j$, $e \rightarrow_{hb}^* e'$ in E', then $e \rightarrow_{hb}^* e'$ in E_j. This can be shown if we show that for any $e, e' \in I_j$, (1) if $e \rightarrow_p e'$ in E', then $e \rightarrow_p e'$ in E_j, and (2) if $e \rightarrow_c e'$ in E', then $e \rightarrow_c e'$ in E_j. We next prove these two facts.

(1) Observe that there is no $e \in R$ such that $e_j = src(e)$. This is because if there is such a e, then $e_j \in W$, but there is no load issued in E after e_j. Therefore, such a e cannot exist. Since E is a trace, E_j is obtained by removing the last event of process $p(e_j)$ from I, and e_j cannot be the src of any event in I_j, E_j is a trace. Since $I_j \subset I'$ and E_j is a trace, for any $e, e' \in I_j$, if $e \rightarrow_p e'$ in E', then $e \rightarrow_p e'$ in E_j.

(2) Let $e_{i1}, e_{i2} \in I_j$ and $e_{i1} \rightarrow_c e_{i2}$ in E'. Since E' is a sequentially consistent trace, $i1 < i2$. Consider the following cases:

(a) Let $src(e_{i2}) = e_{i1}$. In Case 1 above we have already shown that $src(e_{i2}) = src'(e_{i2})$ if $e_{i2} \neq e_j$. Since src_j is a restriction of src to the domain $I \setminus \{e_j\}$, $src_j(e_{i2})$ is also e_{i1}. Therefore, $e_{i1} \rightarrow_c e_{i2}$ in E_j.

(b) Let $(e_{i1}, e_{i2}) \in c'$. Therefore, $a(e_{i1}) = a(e_{i2})$. If $p(e_{i1}) = p(e_{i2})$, then e_{i1} will be committed before e_{i2} because $B_a[p(e_{i1})][a(e_{i1})]$ is a FIFOQueue. If $p(e_{i1}) \neq p(e_{i2})$, then e_{i1} will be committed in line 9 in an iteration $k \leq i2$ and e_{i2} will be committed in an iteration $k > i2$. Therefore, e_{i1} will be committed before e_{i2} in E_j. Therefore, $e_{i1} \rightarrow_c e_{i2}$ in E_j.

(c) Let $(src'(e_{i1}), e_{i2}) \in c'$. Therefore, $a(e_{i1}) = a(e_{i2})$. Let $e = src'(e_{i1})$. Therefore, by case (b), e will be committed before e_{i2} in E_j. By case (a), $e = src_j(e_{i1})$. This implies that $(src_j(e_{i1}), e_{i2}) \in c_j$. Therefore, $e_{i1} \rightarrow_c e_{i2}$ in E_j.

Therefore, we have shown that $e_{i1} \rightarrow_c e_{i2}$ in E_j. From the proof of the above two facts, it follows that for any $e, e' \in I_j$, if $e \rightarrow_{hb}^* e'$ in E', then $e \rightarrow_{hb}^* e'$ in E_j

Since $\mathbf{monitor}_{PSO}$ reports an SC violation in the j^{th} iteration, the conditions at lines 3, 4, and 5 in Figure 5 holds. Therefore, there exists an e and $prev$ in I_j such that $e \rightarrow_{hb}^* prev$ in E_j. Further, $prev \rightarrow_p e_j$. Therefore, $e \rightarrow_{hb}^* e_j$ in E. Since $e \in B_a[p(e)][a(e_i)]$, $a(e) = a(e_j)$, and $p(e) \neq p(e_j)$ and e_j is committed (or issued if it is a load) before e is committed in Figure 9, we have $e_j \rightarrow_c e$. Therefore, we have a $\rightarrow_{hb}^*$ cycle in the trace E.

Theorem 4 *(Soundness of TSO Monitoring) If TSO monitor reports a sequential consistency violation, then there exists a trace $E \in \mathcal{T}_{TSO} \setminus \mathcal{T}_{SC}$ of the program.*

Proof. Let $E' = (I', R', W', L', src', c')$ be an SC trace of some program P and $e_1, \ldots, e_n$ be a linearization of trace E' that respects the $\rightarrow_{hb}^*$ order defined by E'.

Suppose $\mathbf{monitor}_{TSO}$ reports a violation of SC on this linearization in iteration j at line number 7. After printing *Sequential Consistency Violated* at line 7, let $\mathbf{monitor}_{TSO}$ execute the following extra statements shown in Figure 10

```
1   if e_i ∈ L'
2       interlocked_TSO(l(e_i), p(e_i), a(e_i))
3   else if e_i ∈ R'
4       load_TSO(l(e_i), p(e_i), a(e_i))
5   else if e_i ∈ W'
6       store_TSO(l(e_i), p(e_i), a(e_i))
7       while B[p(e_i)].empty()
8           store^c_TSO(p(e_i))
9   foreach p ∈ Proc
10      while B[p].empty()
11          store^c_TSO(p)
```

Fig. 10: Extra statements for TSO monitoring

During this monitoring, $\mathbf{monitor}_{TSO}$ will call a sequence of transitions $\mathbf{store}_{TSO}$, $\mathbf{store}^c_{TSO}$, $\mathbf{load}_{TSO}$, and $\mathbf{interlocked}_{TSO}$. These calls will generate a trace $E = (I, R, W, L, src, c)$. We prove that (1) E is a TSO trace of P and (2) E has a $\rightarrow^*_{hb}$ cycle.

[(1) Proof of E is a TSO trace of P].

Since we call $\mathbf{interlocked}_{TSO}(l(e_i), p(e_i), a(e_i))$, $\mathbf{store}_{TSO}(l(e_i), p(e_i), a(e_i))$, $\mathbf{load}_{TSO}(l(e_i), p(e_i), a(e_i))$ for each e_i in the sequence $e_1, e_2, \ldots, e_j$ if $e_i \in L'$, $e_i \in W'$, and $e_i \in R'$, respectively (see lines 12–17 in Figure 4 and lines 1–8 in Figure 10), I will be the set $\{e_1, e_2, \ldots, e_j\}$, $R = R' \cap I$, $W = W' \cap I$, and $L = L' \cap I$. However, *src* may not be the function src' restricted to the domain R. Let src'' be the function src' restricted to the domain R. Consider the following two cases:

[Case 1] Let $e_{i1} = src'(e_{i2})$ for some $e_{i2} \in R$ such that $e_{i2} \neq e_j$. Since E' is a SC trace of P and $e_1, \ldots, e_n$ is a linearization of the $\rightarrow^*_{hb}$ relation on E', we have $i1 < i2$. Then we can prove that $src'(e_{i2}) = src(e_{i2})$ as follows.

- If $p(e_{i1}) = p(e_{i2})$ and there is no $e_{i3} \in R'$ such that $i1 < i3 < i2$, $a(e_{i3}) = a(e_{i2})$, and $p(e_{i3}) \neq p(e_{i2})$, then the condition at line 8 is not true. Therefore, e_{i1} will be in $B[p(e_{i1})]$ and $src(e_{i2})$ will be e_{i1}, i.e. $src'(e_{i2}) = src(e_{i2})$.
- If $p(e_{i1}) = p(e_{i2})$ and there is a $e_{i3} \in R'$ such that $i1 < i3 < i2$, $i3$ is minimal, $a(e_{i3}) = a(e_{i2})$, and $p(e_{i3}) \neq p(e_{i2})$, then the condition at line 3 is not true because the $i3^{th}$ iteration of $\mathbf{monitor}_{TSO}$ has removed e_{i1} from $B[p(e_{i1})]$. Therefore, e_{i1} will be in $m[a(e_{i1})]$ and $src(e_{i2})$ will be e_{i1}, i.e. $src'(e_{i2}) = src(e_{i2})$.
- If $p(e_{i1}) \neq p(e_{i2})$, then e_{i1} will be in $m[a(e_{i1})]$ and $src(e_{i2})$ will be e_{i1}, i.e. $src'(e_{i2}) = src(e_{i2})$.

– There is no $e_{i3} \in W'$ such that $i1 < i3 < i2$ and $a(e_{i3}) = a(e_{i2})$.

[Case 2] Let $e_{i1} = src'(e_j)$. Since the j^{th} iteration reports SC violation, conditions at lines 3, 5, and 6 hold. Therefore, e_{i1} is in $B[p(e_{i1})]$ and $p(e_j) \neq p(e_{i1})$; hence, $\mathbf{load}_{TSO}$ at line 4 in Figure 10 will not read value stored by e_{i1}, i.e. $src(e_j) \neq e_{i1}$.

Therefore, we showed that src is equal to src'' (recall that src'' is the function src' restricted to the domain R) except for at most one event e_j. Note that e_j is the last event of the process $p(e_j)$ in the trace E. Therefore, by Axiom 1, since E' is a trace of the program P, (I, R, W, L, src'', c'') for some c'' is a trace of the program P. Further by Axiom 2, the trace $E = (I, R, W, L, src, c)$ is a trace of the program P. Since E is generated from the operational model of TSO, E is a TSO trace. Therefore, E is a TSO trace of the program P.

[(2) Proof of E has a $\rightarrow^*_{hb}$ cycle].

Let $I_j = I \setminus \{e_j\}$ and $E \mid_{I_j}$ be $E_j = (I_j, R_j, W_J, L_j, src_j, c_j)$. We want to show that if for any $e, e' \in I_j$, $e \rightarrow^*_{hb} e'$ in E', then $e \rightarrow^*_{hb} e'$ in E_j. This can be shown if we show that for any $e, e' \in I_j$, (1) if $e \rightarrow_p e'$ in E', then $e \rightarrow_p e'$ in E_j, and (2) if $e \rightarrow_c e'$ in E', then $e \rightarrow_c e'$ in E_j. We next prove these two facts.

(1) Observe that there is no $e \in R$ such that $e_j = src(e)$. This is because if there is such a e, then $e_j \in W$, but there is no load issued in E after e_j. Therefore, such a e cannot exist. Since E is a trace, E_j is obtained by removing the last event of process $p(e_j)$ from I, and e_j cannot be the src of any event in I_j, E_j is a trace. Since $I_j \subset I'$ and E_j is a trace, for any $e, e' \in I_j$, if $e \rightarrow_p e'$ in E', then $e \rightarrow_p e'$ in E_j.

(2) Let $e_{i1}, e_{i2} \in I_j$ and $e_{i1} \rightarrow_c e_{i2}$ in E'. Since E' is a sequentially consistent trace, $i1 < i2$. Consider the following cases:

(a) Let $src(e_{i2}) = e_{i1}$. In Case 1 above we have already shown that $src(e_{i2}) = src'(e_{i2})$ if $e_{i2} \neq e_j$. Since src_j is a restriction of src to the domain $I \setminus \{e_j\}$, $src_j(e_{i2})$ is also e_{i1}. Therefore, $e_{i1} \rightarrow_c e_{i2}$ in E_j.

(b) Let $(e_{i1}, e_{i2}) \in c'$. Therefore, $a(e_{i1}) = a(e_{i2})$. If $p(e_{i1}) = p(e_{i2})$, then e_{i1} will be committed before e_{i2} because $B[p(e_{i1})]$ is a FIFOQueue. If $p(e_{i1}) \neq p(e_{i2})$, then e_{i1} will be committed in lines 10 or 11 in an iteration $k \leq i2$ and e_{i2} will be committed in an iteration $k > i2$. Therefore, e_{i1} will be committed before e_{i2} in E_j. Therefore, $e_{i1} \rightarrow_c e_{i2}$ in E_j.

(c) Let $(src'(e_{i1}), e_{i2}) \in c'$. Therefore, $a(e_{i1}) = a(e_{i2})$. Let $e = src'(e_{i1})$. Therefore, by case (b), e will be committed before e_{i2} in E_j. By case (a), $e = src_j(e_{i1})$. This implies that $(src_j(e_{i1}), e_{i2}) \in c_j$. Therefore, $e_{i1} \rightarrow_c e_{i2}$ in E_j.

Therefore, we have shown that $e_{i1} \rightarrow_c e_{i2}$ in E_j. From the proof of the above two facts, it follows that for any $e, e' \in I_j$, if $e \rightarrow^*_{hb} e'$ in E', then $e \rightarrow^*_{hb} e'$ in E_j

Since $\mathbf{monitor}_{TSO}$ reports an SC violation in the j^{th} iteration, the conditions at lines 3, 5, and 6 in Figure 4 holds. Therefore, there exists an e and $prev$ in I_j such that $e \rightarrow^*_{hb} prev$ in E_j. Further, $prev \rightarrow_p e_j$. Therefore, $e \rightarrow^*_{hb} e_j$ in E. Since $e \in B[p(e)]$, $a(e) = a(e_j)$, and $p(e) \neq p(e_j)$ and e_j is committed before e in Figure 10, we have $e_j \rightarrow_c e$. Therefore, we have a $\rightarrow^*_{hb}$ cycle in the trace E.

Theorem 5 *(Completeness of PSO Monitoring) If there exists a trace $E \in \mathcal{T}_{PSO} \setminus \mathcal{T}_{SC}$ of program P, then there exists a sequentially consistent trace E' of P such that PSO monitoring on E' reports a violation of sequential consistency.*

Proof. Along the lines of [3], we define a *relaxed happens-before* relation $\rightarrow_{rhb}$ for a trace E as a subset of the of the $\rightarrow_{hb}$ relation on E:

$$\begin{aligned} \rightarrow_{rhb} \;=\; \rightarrow_{hb} \;&\setminus\; \{(e,e') :\; e \rightarrow_p e',\; e \in W \setminus L,\; e' \in R \setminus L\} \\ &\setminus\; \{(e,e') :\; e \rightarrow_p e',\; e \in W \setminus L,\; e' \in W,\; a(e) \neq a(e')\} \end{aligned}$$

That is, we remove from $\rightarrow_{rhb}$ program-order edges from non-interlocked writes to non-interlocked reads and from non-interlocked writes to writes of different addresses.

Then, let E be (I, R, W, L, src, c) and n be $|I|$. We define events $e_1, \ldots, e_n$ and sets of events $I_0, I_1, \ldots, I_n$ as follows. First, $I_n = I$, the set of all events in trace E. We define each I_j by selecting one event e_{j+1} from I_{j+1} and setting $I_j = I_{j+1} \setminus \{e_{j+1}\}$. Similarly, we define $R_j = R \cap I_j$, $W_j = W \cap I_j$, and $L_j = L \cap I_j$.

Each e_{j+1} is selected as follows. First, if possible, we select an $e_{j+1} \in I_{j+1}$ such that:

(a) Event e_{j+1} is the last event in I_{j+1} issued by its process $p(e_{j+1})$. That is, $\nexists e' \in I_{j+1}.\; e_{j+1} \rightarrow_p e'$.

(b) Event e_{j+1} does not *relaxed* happens-before a read of or an interlocked operation on $a(e_{j+1})$.
Note that this condition implies that e_{j+1} is not the source of a read. That is, $\nexists e \in R_{j+1}.\; src_{j+1}(e) = e_{j+1}$.

(c) Removing event e_{j+1} from E_{j+1} does not remove all happens-before cycles. That is, the $\rightarrow_{hb}$ relation of E_{j+1} has a cycle on $I_{j+1} \setminus e_{j+1}$.

When it is not possible to satisfy condition **(c)**, we can remove any event e_{j+1} satisfying only **(a)** and **(b)**.

We now prove that: (1) there always exists an e_{j+1} satisfying the first two properties, and (2) the resulting restrictions $E\,|_{I_0}, E\,|_{I_1}, \ldots, E\,|_{I_n}$ are all PSO traces. We prove this by induction on j.

Since $I_n = I$, we have that $E\,|_{I_n} = E$ and is a PSO trace.

Suppose $E_{j+1} = E\,|_{I_{j+1}}$ is a PSO trace.

(1) Suppose that the last events of every processor in PSO trace E_{j+1} fail condition **(b)**. Call these events $last_1, \ldots, last_m$.

Because E_{j+1} is a PSO trace, it is output by some run of the PSO automaton. Recall that such a run is a sequence of memory model transitions $\mathbf{store}_{PSO}$, $\mathbf{store}^{c}_{PSO}$, $\mathbf{load}_{PSO}$, and $\mathbf{interlocked}_{PSO}$. During this run, the automaton maps each $e \in I_{j+1}$ to a natural number $C(e)$.

Note that, for any two events e, e' in a trace E from a PSO run, if $e \rightarrow_{rhb} e'$, then we will have $C(e) < C(e')$ for the numbers $C(e)$ and $C(e')$ assigned during the PSO run. There are several cases: **(I)** If $e \rightarrow_c e'$, $e \in W$, and

$e' \in R$, then $a(e') = a(e)$ and either $p(e') \neq p(e)$ or $e \in L$ or $e' \in L$. In any case, $C(e)$ is assigned before $C(e')$. **(II)** If $e \rightarrow_c e'$, $e \in W$ and $e' \in W \setminus R$, then $a(e') = a(e)$ and e is committed before e'. **(III)** If $e \rightarrow_c e'$ and $e \in R \setminus L$, then e' cannot have committed yet when e is issued, because either $src(e) = \perp$ or $src(e) \rightarrow_c e'$. **(IV)** If $e \rightarrow_p e'$ and e is in R, then e is issued before e' is issued or committed, so $C(e) < C(e')$. **(V)** If $e \rightarrow_p e'$ and $e \in W \setminus L$, then e' is a write to the same address. When $e' \in W \setminus L$, then e is buffered before e' in $B[p(e)][a(e)]$, so it is removed and committed first. When $e' \in L$, write e will be removed from $B[p(e)][a(e)]$ by **interlocked**$_{PSO}$ before e' is committed and assigned $C(e')$.

Now, of the events $last_1, \ldots, last_m$, let $last_i$ be the one with largest $C(last_i)$. By our assumption, there is some operation $e \in R_{j+1}$ on $a(last_i)$ such that $last_i \rightarrow^*_{rhb} e$. Let $last_h$ be the last event issued by processor $p(e)$ in E_{j+1}. Either $last_h = e$ or we have $e \rightarrow_p last_h$ and thus $e \rightarrow^*_{rhb} last_h$ because e is a read. Thus, $C(last_i) < C(e) \leq C(last_h)$, contradicting that no $C(last_1), \ldots, C(last_m)$ is larger than $last_i$.

(2) Recall that $E = (I, R, W, L, src, c)$ and E_k is the restriction $E \mid_{I_k} = (I_k, R_k, W_k, L_k, src_k, c_k)$, where $R_k = R \cap I_k$, $W_k = W \cap I_k$, $L_k = L \cap I_k$, $c_k = c \cap W_k \times W_k$, and src_k is the restriction of src to R_k. Note that:

$$\begin{aligned} I_k &= I_k \cap I = I_k \cap (R \cup W) = R_k \cup W_k \\ L_k &= I_k \cap L = I_k \cap (R \cap W) = R_k \cap W_k \end{aligned}$$

Further, for any $(l, p, a, i) \in I_k$ and $1 \leq j < i$, we have $(l, p, a', j) \in I_k$ because condition **(a)** only allows us to remove in each step the final event from any process.
Recall that $E = (I, R, W, L, src, c)$ and tuple E_k is the restriction $E \mid_{I_k} = (I_k, R_k, W_k, L_k, src_k, c_k)$, where $R_k = R \cap I_k$, $W_k = W \cap I_k$, $L_k = L \cap I_k$, $c_k = c \cap W_k \times W_k$, and src_k is the restriction of src to R_k.
Finally, for any $e \in R_k$, we must have that $src(e) \in W_k$, because $src(e) \in W_{k+1}$ and condition **(b)** forbids removing the src of a read from E_{k+1}.
Thus, tuple $E_k = E \mid_{I_k}$ is a PSO trace.

Note also that, by Axiom 1, each $E_k = E \mid_{I_k}$ is a trace of program P.

Then, $E_0 \in \mathcal{T}_{SC}$, because E_0 is the empty trace, and $E_n \in \mathcal{T}_{PSO} \setminus \mathcal{T}_{SC}$. Thus, there is maximal E_j that is in $\mathcal{T}_{SC}$. Let it be E_k. Further, let E' be the sequentially-consistent trace resulting from executing one more instruction in process $p(e_{k+1})$ from trace E_k. That is, E' is trace E_k with one additional event e_{final} with $p(e_{final}) = p(e_{k+1})$. Note that e_{final} must be the same type of event as e_{k+1}—i.e. they are both writes or both reads—although it may have different src or position in the commit order.

We now prove that the PSO monitor algorithm reports a violation on E':

(1) Let $prev$ be the event just before e_{final} in process $p(e_{final})$. Such an event must exist.

Suppose there were no events in process $p(e_{final})$ preceding e_{final}. Then e_{k+1} is the only event in process $p(e_{k+1})$ in E_{k+1}, and thus has no incoming or outgoing $\rightarrow_p$ edges. Thus, the $\rightarrow_{hb}$-cycle containing e_{k+1} contains conflict edges $e' \rightarrow_c e_{k+1} \rightarrow_c e''$, with $e'' \rightarrow^*_{hb} e'$. By condition **(b)** above, event e'' cannot be a read. Thus, it must be the case that $e' \rightarrow_c e''$. But then $e' \rightarrow_c e'' \rightarrow^*_{hb} e'$ is an $\rightarrow_{hb}$-cycle in E', contradicting that E' is SC.

(2) Let *last* denote the last write to $a(e_{final})$ in E_k under the $\rightarrow_{hb}$ relation.
Recall that the $\rightarrow_{hb}$ relation totally orders the writes on any single address in E_k. Thus, *last* is well-defined if there are any writes to $a(e_{k+1})$ in E_k.
There must be such a write. Because e_{k+1} is in an $\rightarrow_{hb}$-cycle in E_{k+1}, there is a conflict edge $e_{k+1} \rightarrow_c e'$ in E_{k+1}. By condition **(b)**, event e' cannot be a read. Thus, e' is a write of $a(e_k) = a(e_{final})$.

(3) It must be the case that $e_{k+1} \rightarrow_c last$ in E_{k+1}.
If not, then e_{k+1} can have no outgoing conflict edges to any write of $a(e_{k+1})$ in E_{k+1}. But, by condition **(b)**, event e_{k+1} can have no outgoing conflict edges to any reads of $a(e_{k+1})$ either. This contradicts that e_{k+1} is in an $\rightarrow_{hb}$-cycle and thus has an outgoing $\rightarrow_c$ edge.

(4) Then, note that $p(last) \neq p(e_{final})$.
If *last* and e_{final} were in the same process, then so would be *last* and e_{k+1}, so $last \rightarrow_p e_{k+1}$ in E_{k+1}. But because *last* and e_{k+1} are operations on the same address, PSO would guarantee that $last \rightarrow_c e_{k+1}$. This contradicts $e_{k+1} \rightarrow_c last$, shown in Point (3) above.

(5) Note also that $last \rightarrow^*_{hb} e_{final}$.
Event e_{final} is obtained by continuing a sequentially-consistent execution of E_k in which *last* has already been issued and committed. Thus, as e_{final} is a read or store of $a(last)$, we have $last \rightarrow^*_c e_{final}$.

(6) Further, we have that $last \rightarrow^*_{hb} prev$.

Let f_1 be the final event issued by processor $p(last)$ in E_k. Recall from Point 4 that $p(last) \neq p(prev)$, and note that it could be that $f_1 = last$. If f_1 relaxed happens-before some other event $r_1 \in R$ on $a(f_1)$ in E_k, then we define f_2 to to be the final event issued by processor $p(r_1)$ in E_k. We similarly define sequences $r_1, \ldots, r_m$ and $f_1, \ldots, f_{m+1}$ of events in E_k where $f_i \rightarrow^*_{rhb} r_i$ in E_k and f_{i+1} is the final event issued by processor $p(r_i)$. The sequence ends when f_{m+1} does not relaxed happens-before any read (or interlocked operation) of $a(f_{m+1})$.

This sequence must be finite, and all of the r_i distinct and all of the f_i distinct, because clearly $f_i \rightarrow^*_{rhb} r_i$ in E_k and either $r_i \rightarrow_p f_{i+1}$ in E_k or $f_{i+1} = r_i$ for all i, and because $\rightarrow^*_{hb}$ is acyclic on the events of E_k.

Suppose f_{m+1} relaxed happens-before a read or interlocked operation e' of $a(f_{m+1})$ in E_{k+1}. The $\rightarrow_{rhb}$ path from f_{m+1} to e' must go through e_{k+1}, as $f_{m+1} \not\rightarrow^*_{rhb} e'$ in E_k. Suppose further that this path does not go through *prev*. Then, either $f_{m+1} \rightarrow^*_{rhb} e'' \rightarrow_c e_{k+1}$ or $f_{m+1} \rightarrow_c e_{k+1}$. But then either $e'' \rightarrow_c last$ in E_k or $f_{m+1} \rightarrow_c last$ in E_k, yielding $\rightarrow_{hb}$-cycle $last \rightarrow^*_{hb} f_{m+1} \rightarrow^*_{hb} last$ in E_k. This is a contradiction, so $last \rightarrow^*_{hb} f_{m+1} \rightarrow^*_{hb} prev$.

Suppose instead that f_{m+1} does *not* relaxed happens-before a read or interlocked operation on $a(f_{m+1})$ in E_{k+1}. Then, f_{m+1} cannot have satisfied condition **(c)** in E_{k+1} (or else we would have removed it instead of e_{k+1}). That is, neither e_{k+1} nor f_{m+1} satisfies condition **(c)** in E_{k+1}, so removing either of e_{k+1} or f_{m+1} would have broken every $\rightarrow_{hb}$-cycle in E_{k+1}. Then, consider some $\rightarrow_{hb}$-cycle $e_{k+1} \rightarrow_c e' \rightarrow^*_{hb} \mathit{prev} \rightarrow_p e_{k+1}$ in E_{k+1}. Event f_{m+1} must also be in this cycle, and thus $f_{m+1} \rightarrow^*_{hb} \mathit{prev}$. Therefore, as $\mathit{last} \rightarrow^*_{hb} f_{m+1}$, we again have $\mathit{last} \rightarrow^*_{hb} \mathit{prev}$.

(7) Note that *last* cannot be an interlocked operation.
This follows from condition **(b)** and because $e_{k+1} \rightarrow_c \mathit{last}$.

(8) Suppose $e \in R \setminus L$ is a read of address $a(\mathit{last})$ in E_k. Then, if $p(e) \neq p(\mathit{last})$, we have that $e \rightarrow^*_{hb} \mathit{last}$ in E'.
Suppose that it is not the case that $e \rightarrow^*_{hb} \mathit{last}$ in E'. Then $\mathit{src}(e)$ is defined and either $\mathit{src}(e) = \mathit{last}$ or $\mathit{last} \rightarrow_c \mathit{src}(e)$ in E' and thus in E_k. But the first alternative contradicts condition **(b)** and the second contradicts that *last* is the last write to $a(\mathit{last})$.

We can now prove that the PSO monitor algorithm will report a violation when run on sequentially-consistent trace E'. The events of E' are presented to $\mathbf{monitor}_{PSO}$ in some linear order $e_1, \ldots, e_{k_1}$ that respects the $\rightarrow^*_{hb}$ order.

Suppose algorithm $\mathbf{monitor}_{PSO}$ has reached event e_{final} without yet reporting a violation. The monitor must have already processed event *last*, because $\mathit{last} \rightarrow^*_{hb} e_{\mathit{final}}$ by Point 5 above.

Further, event *last* must be still be in buffer $B_a[p(\mathit{last})][a(e_{\mathit{final}}]$ at this point. This is because *last* is in $W \setminus L$ by Point 7, and because *last* is removed from this buffer only if there is a later event e which is either: (1) a write or interlocked operation on $a(\mathit{last})$, or (2) a non-interlocked read of $a(\mathit{last})$ in a different process than $p(\mathit{last})$. The first alternative is not possible because, by Point 2, *last* happens-after all other writes to $a(\mathit{last})$ in E', besides possibly e_{final}. The second alternative is not possible because, by Point 8, *last* happens-after all reads of $a(e_{\mathit{final}})$ from different processors, besides possibly e_{final}.

Thus, the condition at Line 3 of $\mathbf{monitor}_{PSO}$ will be satisfied with $e = \mathit{last}$.

The condition at Line 4 is satisfied because Point 1 guarantees the existence of some previous event *prev* in e_{final}'s process.

Finally, the two conditions at Line 5 are satisfies because of Points 4 and 6.

Therefore, algorithm $\mathbf{monitor}_{PSO}$ will report a violation on sequentially-consistent trace E'.

Theorem 6 *(Completeness of TSO Monitoring) If there exists a trace $E \in \mathcal{T}_{TSO} \setminus \mathcal{T}_{SC}$ of program P, then there exists a sequentially consistent trace E' of P such that TSO monitoring on E' reports a violation of sequential consistency.*

Proof. Along the lines of [3], we define a *relaxed happens-before* relation $\rightarrow_{rhb}$ on a trace E in terms of the $\rightarrow_{hb}$ relation on E:

$$\rightarrow_{rhb} \;=\; \rightarrow_{hb} \;\setminus\; \{(e, e') : \; e \rightarrow_p e', \; e \in W \setminus L, \; e' \in R \setminus L\}$$

That is, we remove from $\rightarrow_{rhb}$ program-order edges from non-interlocked writes to non-interlocked reads.

Then, let E be (I, R, W, L, src, c) and n be $|I|$. We define events $e_1, \ldots, e_n$ and sets of events $I_0, I_1, \ldots, I_n$ as follows. First, $I_n = I$, the set of all events in trace E. We define each I_j by selecting one event e_{j+1} from I_{j+1} and setting $I_j = I_{j+1} \backslash \{e_{j+1}\}$. Similarly, we define $R_j = R \cap I_j$, $W_j = W \cap I_j$, and $L_j = L \cap I_j$.

Each e_{j+1} is selected as follows. First, if possible, we select an $e_{j+1} \in I_{j+1}$ such that:

- **(a)** Event e_{j+1} is the last event in I_{j+1} issued by its process $p(e_{j+1})$. That is, $\nexists e' \in I_{j+1}.\ e_{j+1} \rightarrow_p e'$.
- **(b)** Event e_{j+1} is not the source of a read. That is, $\nexists e \in R_{j+1}.\ src_{j+1}(e) = e_{j+1}$.
- **(c)** Removing event e_{j+1} from E_{j+1} does not remove all happens-before cycles. That is, the $\rightarrow_{hb}$ relation of E_{j+1} has a cycle on $I_{j+1} \setminus e_{j+1}$.
- **(d)** Event e_{j+1} does not relaxed happens-before a read of $a(e_{j+1})$ or an interlocked operation of any address.
- **(e)** Let *last* denote the last write to $a(e_{j+1})$ in $E_{j+1} \backslash e_{j+1}$ under the $\rightarrow_{hb}$ relation. *last*, if it exists, does not relaxed happens-before any operation e such that $a(e) \neq a(last)$ and $p(e) \neq p(last)$. *last* does not relaxed happens-before an interlocked operation on any address on the same thread.

When it is not possible to satisfy conditions **(d)** and **(e)**, we can remove any event e_{j+1} satisfying only **(a)**, **(b)** and **(c)**.

When it is not possible to satisfy condition **(c)**, we can remove any event e_{j+1} satisfying only **(a)**, **(b)**, **(d)** and **(e)**.

We now prove that: (1) there always exists an e_{j+1} satisfying properties **(a)**, **(b)**, **(c)** or **(a)**, **(b)**, **(d)**, **(e)**, and (2) the resulting restrictions $E\ |_{I_0}, E\ |_{I_1}, \ldots, E\ |_{I_n}$ are all TSO traces. We prove this by induction on j.

Since $I_n = I$, we have that $E\ |_{I_n} = E$ and is a TSO trace.

Suppose that E_{j+1} is a TSO trace.

(1) Suppose that there does not exist an event in E_{j+1} satisfying either conditions **(a)**, **(b)** and **(c)** or conditions **(a)**, **(b)**, **(d)** and **(e)**. This is equivalent to saying that all processors' last events in E_{j+1} fail condition **(b)** or conditions **(c)** and **(d)** or conditions **(c)** and **(e)**. Call these events $last_1, \ldots, last_m$.

Because E_{j+1} is a TSO trace, it is output by some run of the TSO automaton. Recall that such a run is a sequence of memory model transitions $\mathbf{store}_{TSO}$, $\mathbf{store}^c_{TSO}$, $\mathbf{load}_{TSO}$, and $\mathbf{interlocked}_{TSO}$. During this run, the automaton maps each $e \in I_{j+1}$ to a natural number $C(e)$.

Note that, for any two events e, e' in a trace E from a TSO run, if $e \rightarrow_{rhb} e'$, then we will have $C(e) < C(e')$ for the numbers $C(e)$ and $C(e')$ assigned during the TSO run. There are several cases: **(I)** If $e \rightarrow_c e'$, $e \in W$, and $e' \in R$, then $a(e') = a(e)$ and either $p(e') \neq p(e)$ or $e \in L$ or $e' \in L$. In any case, $C(e)$ is assigned before $C(e')$. **(II)** If $e \rightarrow_c e'$, $e \in W$ and $e' \in W \setminus R$, then $a(e') = a(e)$ and e is committed before e'. **(III)** If $e \rightarrow_c e'$ and $e \in R \setminus L$, then e' cannot have committed yet when e is issued, because

either $src(e) = \perp$ or $src(e) \rightarrow_c e'$. **(IV)** If $e \rightarrow_p e'$ and e is in R, then e is issued before e' is issued or committed, so $C(e) < C(e')$. **(V)** Suppose $e \rightarrow_p e'$ and $e \in W \setminus L$. When $e' \in W \setminus L$, then e is buffered before e' in $B[p(e)]$, so it is removed and committed first. When $e' \in L$, write e will be removed from $B[p(e)]$ by $\mathbf{interlocked}_{TSO}$ before e' is committed and assigned $C(e')$.

Thus, it follows that that $\rightarrow_{rhb}$ is acyclic on TSO traces, including E_{j+1}.

By our assumption, for each event $last_i$ from $last_1, \ldots, last_m$ there are three cases: (1) $last_i$ is the source of a read (2) there is some operation $e \in R_{j+1}$ on $a(last_i)$ or in L_{j+1}, such that $last_i \rightarrow^*_{rhb} e$ and removing e_{j+1} removes all happens-before cycles or (3) if $last$ is the last write to $a(last_i)$ in E_{j+1} there is some operation $e \in I_{j+1}$ on $a(e) \neq a(last)$ in $p(e) \neq p(last)$ and $p(e) \neq p(e_{j+1})$, such that $last \rightarrow^*_{rhb} e$ and removing e_{j+1} removes all happens-before cycles.
Let F_h be the subset of $last_1, \ldots, last_m$ such that all events in F_h fail condition **(c)**. If any of the events in F_h is removed from E_{j+1} then all happens-before cycles are removed. It follows then that all events in F_h belong to the same same set of happens-before cycles.
Events in $\{last_1, \ldots, last_m\} \setminus F_h$ do not fail condition **(c)** hence by our assumption they should fail condition **(b)**. Therefore for all $last_i$ not in F_h it is true that $last_i$ is a write operation and is the source of a read in E_{j+1}.
We pick now one of the cycles that events in F_h belong to. We order the events of $F_h : f_1, \ldots, f_h$ in a way such that $f_i \rightarrow^*_{hb} f_j$ if $i < j$ and f_h is the event with the largest commit number C in F_h. We examine event f_h. Event f_h fails condition **(c)** and it also fails condition **(d)** or **(e)**:

(A) Suppose it fails condition **(d)**. Then there exists a read of address $a(f_h)$ or interlocked operation w_h such that $f_h \rightarrow^*_{rhb} w_h$. Let f_{h+1} be the last event in processor $p(w_h)$. Since $w_h \rightarrow_p f_{h+1}$ and w_h is not a write, it is true that $w_h \rightarrow_{rhb} f_{h+1}$. Therefore f_h relaxed happens-before f_{h+1}, so $C(f_h) < C(f_{h+1})$ and $f_{h+1} \notin F_h$ because f_h was defined to be the event with the largest commit number in F_h. Since f_{h+1} fails condition **(b)**, let w_{h+1} be the read whose source is f_{h+1}. Let f_{h+2} be the last event in processor $p(w_{h+1})$. Since w_{h+1} is a read, $w_{h+1} \rightarrow_{rhb} f_{h+2}$ hence $f_h \rightarrow^*_{rhb} f_{h+2}$. For the same reason as before $f_{h+2} \notin F_h$ and f_{h+2} must fail condition **(b)**. We continue this construction to build a sequence of events $f_{h+1}, w_{h+1}, f_{h+2}, \ldots w_{h+i-1}, f_{h+i}$ where events $f_{h+i} \notin F_h$ and f_{h+i} fail condition **(b)**. Since there is a finite number of processors there must exist k such that $f_{h+k} = f_{h+j}$ for $1 < j < k$. However that would create a relaxed happens-before cycle in E_{j+1} so we reached a contradiction.

(B) Suppose f_h fails condition **(e)**. Let $last$ be the last write to address $a(f_h)$. Then $last$ relaxed happens-before an operation e such that $p(e) \neq p(last)$ and $p(e) \neq p(f_h)$. Let f'_1 be the last event in processor $p(last)$ and f''_1 the last event in processor $p(e)$. If f'_1 is any of the events $f_2, \ldots f_{h-1}$ then the happens-before cycle $f_h \rightarrow_c last \rightarrow_p f'_1 \rightarrow^*_{hb} f_h$ does not include

event f_1 and is not removed when f_1 is removed which contradicts the assumption that f_1 fails condition **(c)**. In the same way if f_1'' cannot be any of the events $f_2, \dots f_{h-1}$ because of the happens-before cycle $f_h \rightarrow_c last \rightarrow^*_{rhb} e \rightarrow_p f_1'' \rightarrow^*_{hb} f_h$. Therefore f_1' and f_1'' are either event f_1 or are not in F_h and thus fail condition **(b)** and are sources of loads. Suppose f_1' is a source of a load and not f_1. Then f_1' is a write and $last \rightarrow_{rhb} f_1'$. In the same way as in (1) we construct a sequence of events $f_1', w_1', f_2', \dots, w_{k-1}', f_k'$ where f_i is the source of w_i' and $f_1' \rightarrow_{rhb} f_k'$. Since the number of processors is limited the sequence will have to reach one of the events in $f_1, \dots f_{h-1}$. It cannot reach f_h or any of the last events already in the sequence because that would form a relaxed happens-before cycle. But since $f_1' \rightarrow^*_{rhb} f_k'$ and $f_h \rightarrow^*_{rhb} f_1'$, then f_h would have to relaxed happen-before the event f_i for $1 \leq i < h$ that the sequence reaches. Therefore it would be $C(f_h) < C(f_i)$ which contradicts the fact that f_h has the highest commit number in $f_1, \dots, f_h$. Thus f_1' cannot be a source of a load and has to be f_1. Following the same reasoning we prove that for f_1'' if it is a source of a read and not f_1 then $f_h \rightarrow^*_{rhb} f_1''$ and again we reach a contradiction. Therefore, both f_1' and f_1'' have to be f_1 which cannot be true because they are disctinct events. The initial assumption that f_h fails condition **(e)** is therefore wrong.

In (A) and (B) we proved that f_h cannot fail **(d)** or **(e)**, we reach a contradiction assuming that all events in $last_1, \dots, last_m$ fail conditions **(b)** or **(c)** and **(d)** or **(c)** and **(e)**.

(2) Recall that $E = (I, R, W, L, src, c)$ and E_k is the restriction $E \mid_{I_k} = (I_k, R_k, W_k, L_k, src_k, c_k)$, where $R_k = R \cap I_k$, $W_k = W \cap I_k$, $L_k = L \cap I_k$, $c_k = c \cap W_k \times W_k$, and src_k is the restriction of src to R_k. Note that:

$$\begin{aligned} I_k &= I_k \cap I = I_k \cap (R \cup W) = R_k \cup W_k \\ L_k &= I_k \cap L = I_k \cap (R \cap W) = R_k \cap W_k \end{aligned}$$

Further, for any $(l, p, a, i) \in I_k$ and $1 \leq j < i$, we have $(l, p, a', j) \in I_k$ because condition **(a)** only allows us to remove in each step the final event from any process.

Finally, for any $e \in R_k$, we must have that $src_k(e) = src(e) \in W_k$. Suppose $src(e) \notin W_k$. Because E_{k+1} is a TSO trace, we have $src_{k+1}(e) = src(e) \in W_{k+1}$. Thus, we must have selected $e_{k+1} = src(e)$ to remove. But this contradicts condition **(b)**.

Thus, tuple $E_k = E \mid_{I_k}$ is a TSO trace.

Note also that, by Axiom 1, each trace $E_k = E \mid_{I_k}$ is a trace of program P.

Then, $E_0 \in \mathcal{T}_{SC}$, because E_0 is the empty trace, and $E_n \in \mathcal{T}_{TSO} \setminus \mathcal{T}_{SC}$. Thus, there is maximal E_j that is in $\mathcal{T}_{SC}$. Let it be E_k. E_{k+1} then is not in $\mathcal{T}_{SC}$ and thus contains a happens-before cycle. Event e_{k+1}, when removed from E_{k+1}, removes all happens-before cycles and thus fails condition **(c)**. Therefore, in E_{k+1} it was not possible to select an event that satisfies conditions **(a)**, **(b)** and **(c)** and e_{k+1} must satisfy conditions **(a)**, **(b)**, **(d)** and **(e)**.

Further, let E' be the sequentially-consistent trace resulting from executing one more instruction in process $p(e_{k+1})$ from trace E_k. That is, E' is trace E_k with one additional event e_{final} with $p(e_{final}) = p(e_{k+1})$. Note that e_{final} must be the same type of event as e_{k+1}—i.e. they are both writes or both loads—although it may have different *src* or position in the commit order.

We now prove that the TSO monitor algorithm reports a violation on E':

(1) Note that e_{k+1} can have no outgoing conflict edges to a read $e \in R$ in E_{k+1}. If it were the case that $e_{k+1} \rightarrow_c e$ in E_{k+1} for $e \in R$ and $p(e) \neq p(e_{k+1})$, then we would have $e_{k+1} \rightarrow_{rhb} e \rightarrow_{rhb} e'$, where e' is the last event in process $p(e)$. But this contradicts condition **(d)** above.

(2) Let *prev* be the event just before e_{final} in process $p(e_{final})$. Such an event must exist.

Suppose there were no events in process $p(e_{final})$ preceding e_{final}. Then e_{k+1} is the only event in process $p(e_{k+1})$ in E_{k+1}, and thus has no incoming or outgoing $\rightarrow_p$ edges. Thus, the $\rightarrow_{hb}$-cycle containing e_{k+1} contains conflict edges $e' \rightarrow_c e_{k+1} \rightarrow_c e''$, with $e'' \rightarrow^*_{hb} e'$. By Point 1, event e'' cannot be a read. Thus, it must be the case that $e' \rightarrow_c e''$. But then $e' \rightarrow_c e'' \rightarrow^*_{hb} e'$ is an $\rightarrow_{hb}$-cycle in E_k, contradicting that $E_k \in \mathcal{T}_{SC}$.

(3) Let *last* denote the last write to $a(e_{final})$ in E_k under the $\rightarrow_{hb}$ relation.

Recall that, because E_k is SC, relation $\rightarrow_{hb}$ is acyclic on E_k, and thus $\rightarrow_{hb}$ totally orders the writes on any single address in E_k. Thus, *last* is well-defined if there are any writes to $a(e_{k+1})$ in E_k.

There must be such a write. Because e_{k+1} is in an $\rightarrow_{hb}$-cycle in E_{k+1}, there is a conflict edge $e_{k+1} \rightarrow_c e'$ in E_{k+1}. By Point 1, event e' cannot be a read. Thus, e' is a write of $a(e_k) = a(e_{final})$.

(4) It must be the case that $e_{k+1} \rightarrow_c last$ in E_{k+1}.

Consider the write e' such that $e_{k+1} \rightarrow_c e'$ in the proof of Point 3 above. Clearly $e' \rightarrow_c last$. Thus, as e' and *last* are both writes, $e_{k+1} \rightarrow_c last$.

(5) Then, note that $p(last) \neq p(e_{final})$.

If *last* and e_{final} were in the same process, then so would be *last* and e_{k+1}, so $last \rightarrow_p e_{k+1}$ in E_{k+1}. But because *last* and e_{k+1} are operations on the same address, TSO would guarantee that $last \rightarrow_c e_{k+1}$. This contradicts $e_{k+1} \rightarrow_c last$, shown in Point 4 above.

(6) We have that $last \rightarrow^*_{hb} prev$.

Let f_1 be the final event issued by processor $p(last)$ in E_k. Recall from Point 5 that $p(last) \neq p(prev)$, and note that it could be that $f_1 = last$. Let w_1 be an event that would cause f_1 to fail conditions **(d)** and **(e)**. Then we define f_2 to be the final event issued by processor $p(w_1)$ in E_k. We similarly define sequences $w_1, \ldots, w_m$ and $f_1, \ldots, f_{m+1}$ of events in E_k where $f_i \rightarrow^*_{rhb} w_i$ in E_k and f_{i+1} is the final event issued by processor $p(w_i)$. The sequence ends when f_{m+1} does not fail neither condition **(d)** nor **(e)**.

This sequence must be finite, and all of the w_i distinct and all of the f_i distinct, because clearly $f_i \rightarrow^*_{rhb} w_i$ in E_k and either $w_i \rightarrow_p f_{i+1}$ in E_k or $f_{i+1} = w_i$ for all i, and because $\rightarrow^*_{hb}$ is acyclic on the events of E_k.

Suppose f_{m+1} fails condition **(d)** in E_{k+1}, i.e. f_{m+1} relaxed happens-before a read or interlocked operation e' of $a(f_{m+1})$ in E_{k+1}. The $\rightarrow_{rhb}$ path from f_{m+1} to e' must go through e_{k+1}, as $f_{m+1} \not\rightarrow^*_{rhb} e'$ in E_k. Suppose further that this path does not go through *prev*. Then, either $f_{m+1} \rightarrow^*_{rhb} e'' \rightarrow_c e_{k+1}$ or $f_{m+1} \rightarrow_c e_{k+1}$. But then either $e'' \rightarrow_c last$ in E_k or $f_{m+1} \rightarrow_c last$ in E_k, yielding $\rightarrow_{hb}$-cycle $last \rightarrow^*_{hb} f_{m+1} \rightarrow^*_{hb} last$ in E_k. This is a contradiction, so $last \rightarrow^*_{hb} f_{m+1} \rightarrow^*_{hb} prev$.

Suppose f_{m+1} fails condition **(e)** in E_{k+1}, i.e. the last write e on $a(f_{m+1})$ under the $\rightarrow_{hb}$ relation relaxed happens-before an operation e' of $a(e') \neq a(f_{m+1})$ in E_{k+1}. The $\rightarrow_{rhb}$ path from e to e' must go through e_{k+1}, as $e \not\rightarrow^*_{rhb} e'$ in E_k. Suppose further that this path does not go through *prev*. Then, either $e \rightarrow^*_{rhb} e'' \rightarrow_c e_{k+1}$ or $e \rightarrow_c e_{k+1}$. But then either $e'' \rightarrow_c last$ in E_k or $e \rightarrow_c last$ in E_k, yielding $\rightarrow_{hb}$-cycle $last \rightarrow^*_{hb} e \rightarrow^*_{hb} last$ in E_k. This is a contradiction, so $last \rightarrow^*_{hb} e \rightarrow^*_{hb} prev$ and since $last \rightarrow^*_{hb} f_{m+1}$ and $f_{m+1} \rightarrow^*_{hb} e$ we get $last \rightarrow^*_{hb} f_{m+1} \rightarrow^*_{hb} prev$

Suppose instead that f_{m+1} does *not* fail conditions **(d)** or **(e)** in E_{k+1}. Then, f_{m+1} cannot have satisfied condition **(c)** in E_{k+1} (or else we would have removed it instead of e_{k+1}). That is, neither e_{k+1} nor f_{m+1} satisfies condition **(c)** in E_{k+1}, so removing either of e_{k+1} or f_{m+1} would have broken every $\rightarrow_{hb}$-cycle in E_{k+1}. Then, consider some $\rightarrow_{hb}$-cycle $e_{k+1} \rightarrow_c e' \rightarrow^*_{hb} prev \rightarrow_p e_{k+1}$ in E_{k+1}. Event f_{m+1} must also be in this cycle, and thus $f_{m+1} \rightarrow^*_{hb} prev$. Therefore, as $last \rightarrow^*_{hb} f_{m+1}$, we again have $last \rightarrow^*_{hb} prev$.

We can now prove that the TSO monitor algorithm will report a violation when run on sequentially-consistent trace E'. The events of E' are presented to $\mathbf{monitor}_{TSO}$ in some linear order $e_1, \ldots, e_{k_1}$ that respects the $\rightarrow^*_{hb}$ order.

Suppose algorithm $\mathbf{monitor}_{TSO}$ has reached event e_{final} without yet reporting a violation. The monitor must have already processed event *last*, because $last \rightarrow^*_{hb} e_{final}$ by Point 5 above.

Further, event *last* must be still be in buffer $B[p(last)]$ at this point. Event *last* is flushed only if either: (1) we reach Line 9 with $e = last$, (2) we reach Line 9 with e a write of any address in $p(last)$ after *last*, or (3) we reach Line 13 with $p(e_i) = p(last)$.

In case (1), $a(e_i) = a(last)$ from condition in line 3 and $p(e_i) \neq p(last)$ from condition in line 8. If e_i is a write, this contradicts *last* being the last write to address $a(last)$. If e_i is a read then $last \rightarrow_{rhb} e_i$. But $e_{k+1} \rightarrow_{rhb} last$ therefore $e_{k+1} \rightarrow^*_{rhb} e_i$ and that contradicts condition **(d)**.

In case (2), *last* and e are writes in the same processor therefore $last \rightarrow_{rhb} e$. Further, if e_i is a write then since $a(e) = a(e_i)$, $e \rightarrow_c e_i$ and $e \rightarrow_{rhb} e_i$. If e_i is a read then suppose it is not $e \rightarrow^*_{rhb} e_i$. e cannot be the source of e_i. There must exist write e_j such that $e_j = src(e_i)$. It cannot be $e \rightarrow_c e_j$ because in that case $e \rightarrow^*_{rhb} e_i$. It has to be $e_j \rightarrow_c e$ and hence $e_i \rightarrow_c e \Rightarrow e_i \rightarrow_{hb} e$.

But this contradicts the fact that e_i appears in the monitor algorithm after e so our assumption that it is not $e \rightarrow^*_{rhb} e_i$ has to be wrong. So, in both cases, e_i is a read or a write, we have that $e \rightarrow^*_{rhb} e_i$ and since $last \rightarrow_{rhb} e$, it will be $last \rightarrow^*_{rhb} e_i$. This contradicts condition **(e)**.

Case (3) is not possible because *last* by point 2 happens-after all other writes to $a(last)$ besides possibly e_{final}.

Therefore, algorithm $\mathbf{monitor}_{TSO}$ will report a violation on sequentially-consistent trace E'.

Theorem 7 *(Complexity of TSO/PSO Monitoring) Time complexity of both TSO/PSO monitoring is $O(|Proc|.|E|)$, where $|E|$ is the length of the trace and $|Proc|$ is the number of processes.*

Proof. Both $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ run for n iterations, where $n = |E|$. At the time of monitoring, we associate a vector clock with each event so that we can decide if $e \rightarrow^*_{hb} prev$ in line 6 of $\mathbf{monitor}_{TSO}$ and line 5 of $\mathbf{monitor}_{PSO}$. Computing vector clock for each event has a run-time complexity of $O(|Proc|)$. All the buffers in both $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ have at most $|E|$ distinct pending stores throughout the algorithm. Therefore, the number of cumulative iterations of the loops at line 9 of $\mathbf{monitor}_{TSO}$ and line 8 of $\mathbf{monitor}_{PSO}$ is $|E|$. The lookup of e in both $\mathbf{monitor}_{TSO}$ and $\mathbf{monitor}_{PSO}$ at line 3 can be done in constant time. This is because by Lemma 1 and Lemma 2, pending stores for an address a can be present in only one buffer. Therefore, one can maintain an array indexed by addresses where each element of the array points to the position of the last pending store for the address in the buffer. Therefore, the overall complexity of both monitoring algorithms is $O(|Proc|.|E|)$.