# Maximal Causality Reduction for TSO and PSO

Shiyou Huang
Texas A&M University, USA
huangsy@tamu.edu

Jeff Huang
Texas A&M University, USA
jeff@cse.tamu.edu

## Abstract

Verifying concurrent programs is challenging due to the exponentially large thread interleaving space. The problem is exacerbated by relaxed memory models such as Total Store Order (TSO) and Partial Store Order (PSO) which further explode the interleaving space by reordering instructions. A recent advance, Maximal Causality Reduction (MCR), has shown great promise to improve verification effectiveness by maximally reducing redundant explorations. However, the original MCR only works for the Sequential Consistency (SC) memory model, but not for TSO and PSO. In this paper, we develop novel extensions to MCR by solving two key problems under TSO and PSO: 1) generating interleavings that can reach new states by encoding the operational semantics of TSO and PSO with first-order logical constraints and solving them with SMT solvers, and 2) enforcing TSO and PSO interleavings by developing novel replay algorithms that allow executions out of the program order. We show that our approach successfully enables MCR to effectively explore TSO and PSO interleavings. We have compared our approach with a recent Dynamic Partial Order Reduction (DPOR) algorithm for TSO and PSO and a SAT-based stateless model checking approach. Our results show that our approach is much more effective than the other approaches for both state-space exploration and bug finding – on average it explores 5-10X fewer executions and finds many bugs that the other tools cannot find.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification–Model Checking

***General Terms*** Algorithms, Design, Verification

***Keywords*** Maximal Causality Reduction, Model Checking, TSO, PSO

*OOPSLA'16*, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984025

## 1. INTRODUCTION

Verifying concurrent programs has been a long-standing challenge due to state-space explosion caused by the huge thread interleaving space. It is known that *sequential consistency* (SC) [24] is the most intuitive memory model, under which operations by different threads can interleave but those by the same thread should always follow the program order. It is challenging enough to verify concurrent programs under SC, because the number of different interleavings grows exponentially with the number of threads and the length of program execution. What makes things worse is that to achieve better performance, most contemporary multiprocessors implement *relaxed memory models*, such as *Total Store Order* (TSO) and *Partial Store Order* (PSO) [5, 33]. For TSO and PSO, the verification problem is more challenging because operations by the same thread may no longer follow the program order. For instance, under TSO, a write and a following read by the same thread can be re-ordered if they access different memory locations, and under PSO, which is a further relaxation of TSO, two writes by the same thread can be re-ordered if they target different locations.

The ability to re-order operations from the same thread under TSO and PSO significantly explodes the state-space over SC. Consider $M$ concurrent threads each executing $N_i$ operations where $i$=$1, 2, \dots, M$. The total number of interleavings under SC can be calculated by the formula $\prod_{i=1}^{M}\binom{\sum_{j=i}^{M} N_j}{N_i}$ [27], and that of allowing the reordering of operations can be calculated by the formula $(\prod_{i=1}^{M} N_i)!$ (*i.e.*, the number of permutations of all operations). Consider only four threads and four operations each ($M$=$N_i$=4). The number of interleavings under SC is $6 * 10^7$, whereas the number of permutations is $2 * 10^{13}$, which is 300,000 times larger.

Consequently, bugs may only occur under TSO or PSO but not SC, and those bugs are much more difficult to detect. Figure 1 shows a real bug extracted from a large program (with over 40K lines of code) running on an electron microscope [2]. The program runs safely under SC and TSO. However, an error (lines 15-17 of Figure 1) is triggered when it runs under PSO and unfortunately caused a loss of $12 million of equipment. The root cause of the error is that the

```
class A {
    static Point currentPos = new Point(1,2);
    static class Point {
        int x;
        int y;
        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
    public static void main(String[] args) {
        new Thread() {
            void f(Point p) {
                synchronized(this) {}
                if (p.x+1 != p.y) {
                    System.out.println(p.x+" "+p.y);
                    System.exit(1);
                }
            }
            @Override
            public void run() {
                while (currentPos == null);
                while (true)
                    f(currentPos);
            }
        }.start();
        while (true){
           currentPos =
          new Point(currentPos.x+1, currentPos.y+1);
        }
    }
}
```

**Figure 1:** A real PSO bug in an electron microscope software [2]. This bug caused a \$12 million loss of equipment.

write to the object *curPosition* can happen before the write to the field of the object, which is allowed under PSO. What is worse is that this error can hardly be reproduced. On average, the error appears only once in every $500,000$ loop iterations of the program[1].

A state-of-the-art approach for verifying concurrent programs is model checking [12]. Model checking techniques can be divided into two categories: stateful and stateless, both striving to explore state-space effectively by reducing redundant explorations. Stateful model checking techniques [20, 25, 26, 37] store abstract states at runtime to help avoid redundant explorations. In contrast, *stateless model checking* (which we refer to as SMC in this paper) techniques explore state-space systematically by driving concrete program executions via a dynamic scheduler without storing any states. Since the pioneering work of VeriSoft [17, 18] and CHESS [32], SMC has been successfully applied in real-world programs and has found many deep bugs through optimization techniques such as *partial order reduction* (POR) [16, 19] and context-bounding [32].

A key challenge in SMC is how to avoid redundant explorations of the same program state. Although POR reduces redundancy by characterizing distinct interleavings (*i.e.*, Mazurkiewicz traces [29]) with *happens-before*, it is also limited by *happens-before* and cannot reduce redundant interleavings that have different *happens-before* relations. To maximally reduce redundancy, recently, we developed a new technique called *Maximal Causality Reduction* (MCR) [21]. By taking the value of reads and writes into consideration and by exploiting the maximal causality between redundant executions that lead to equivalent states, MCR ensures that every explored execution reaches a distinct program state. In this way, MCR minimizes the number of executions that must be explored to verify concurrent programs. MCR achieves a significant advance in SMC and has shown better scalability and bug finding capability than POR and context bounding. However, the MCR technique [21] is limited to SC only, and it does not work for TSO and PSO.

Motivated by MCR, we have developed a new technique that realizes MCR for TSO and PSO by solving the following two technical challenges:

1. Soundly encoding the semantics of TSO and PSO (specifically the write-to-read and write-to-write reorderings) by relaxing the SC constraints in MCR [21].
2. Deterministically replaying TSO and PSO interleavings for concurrent programs.

From a high-level perspective, we decompose the *must-happen-before* constraints that are specialized for SC in MCR into two parts: $\Phi_{mem}$ and $\Phi_{sync}$, where $\Phi_{mem}$ captures the reordering semantics allowed by different memory models, and $\Phi_{sync}$ the *happens-before* constraints entailed by synchronizations. Under TSO and PSO, the constraint $\Phi_{mem}$ allows reads and writes to be re-ordered while respecting the semantics of the memory models with store buffers (FIFO queues). We assign one buffer to each thread for TSO and multiple for PSO (each corresponds to a dynamic memory location) to achieve the reordering. The other constraints remain the same as that in MCR. By invoking an off-the-shelf SMT solver to solve the constraints, new interleavings are generated and are used to replay the program to explore new states. We further design a novel algorithm to deterministically replay concurrent programs under TSO and PSO, where operations in the generated interleavings can be re-ordered (*i.e.*, does not follow the program order). The key insight of our algorithm is to decide when to buffer a write and when to flush it into the main memory by comparing the memory location of the executed operation with that of the operation given in the generated interleaving.

We have implemented our new algorithms in MCR and evaluated them with a collection of popular multithreaded benchmarks and real applications. We have also compared our technique with two recent SMC techniques for TSO and PSO: the DPOR algorithm by Zhang [38] and the SATCheck[2] technique by Demsky and Lam [15]. Our experimental results show that MCR with our approach is significantly more effective than DPOR and SATCheck for state-space exploration: it takes 5X fewer executions than DPOR for TSO and 10X fewer for PSO on average, whereas SATCheck misses states. Moreover, our approach is much more effective than the other

[1] Interestingly, our approach takes only three runs to find this PSO bug.

[2] SATCheck works for SC and TSO only.

two approaches for finding TSO and PSO bugs: it typically takes only a few executions to reveal a known error, whereas DPOR and SATCheck either take 3-5X more executions or fail to find the error.

This paper makes the following contributions:

- We present a new technique based on maximal causality reduction (MCR) for effectively verifying concurrent programs under TSO and PSO.
- We formalize TSO and PSO reordering semantics with new MCR constraints and design novel replay algorithms to enforce interleavings under both TSO and PSO.
- We evaluate our technique on popular benchmarks and real applications and our results show significant improvements over the recent approaches.

The rest of the paper is organized as follows: Section 2 introduces necessary background about MCR; Section 3 presents the semantics of TSO and PSO; Section 4 presents our approach for encoding the TSO and PSO constraints and new algorithms for replaying generated interleavings under TSO and PSO. Section 5 presents a detailed case study of our technique on the real bug in Figure 1; Section 6 reports our experimental results; Section 7 discusses related work and Section 8 concludes this paper.

## 2. Maximal Causality Reduction

In this section, we review the key ideas of MCR and illustrate how MCR works with a simple example. We also identify its limitation with respect to TSO and PSO.

### 2.1 Maximal Causal Model

A fundamental concept underpinning MCR is the Maximal Causality Model (MCM) [22, 35], which takes as input an observed execution trace of a multithreaded program and captures the largest set of feasible traces that can be inferred from the observed trace. In MCM, multithreaded programs $\mathcal{P}$ are abstracted as the prefix-closed sets of finite traces, called $\mathcal{P}$-feasible traces, that $\mathcal{P}$ can produce when completely or partially executed. A trace is abstracted as a sequence of events, which are operations performed by threads on concurrent objects. The following common types of events are considered in MCM:

- ***begin(t)/end(t)***: the first/last event of thread $t$;
- ***read(t, x, v)/write(t, x, v)***: read/write $x$ with value $v$;
- ***lock(t,l)/unlock(t,l)***: acquire/release a lock $l$;
- ***fork(t,t′)***: fork a new thread $t'$;
- ***join(t,t′)***: block until thread $t'$ terminates.

There are two important properties held by the sets of $\mathcal{P}$-feasible traces: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible. The latter means the execution of a concurrent operation is only determined by the previous events in the same thread. These two axioms allow us to associate any consistent trace $\tau$ with a maximal set of traces $\mathsf{MaxCausal}(\tau)$, which comprises precisely the traces that can be generated by any program that can generate $\tau$.

It is shown in [22, 35] that $\mathsf{MaxCausal}(\tau)$ is both sound and maximal: any program that can generate $\tau$ can also generate all traces in $\mathsf{MaxCausal}(\tau)$, and for any trace $\tau'$ not in $\mathsf{MaxCausal}(\tau)$ there exists a program generating $\tau$ which cannot generate $\tau'$.

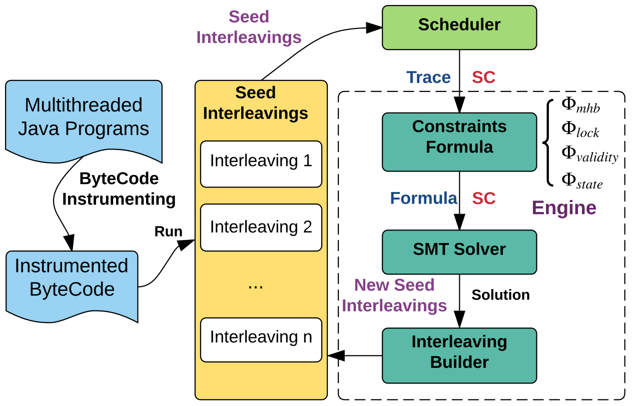


**Figure 2:** MCR workflow under SC

### 2.2 MCR Workflow

As illustrated in Figure 2, given a closed multithreaded program with a fixed input, MCR systematically explores all unique interleavings of the program in a closed loop, with each explored interleaving covering a unique program state. Initially, MCR executes the program following a random interleaving and generates an initial trace $\tau$. Then $\tau$ is used to compute the maximal casual traces $\mathsf{MaxCausal}(\tau)$, and from which MCR generates new *seed interleavings* (if there exist any). Each seed interleaving is produced by encoding $\mathsf{MaxCausal}(\tau)$ together with a *new state* constraint over a read event in $\tau$ enforcing it to read a new value, such that the seed interleaving will drive the program to reach a new state (*i.e.*, at least one read will read a new value). The seed interleavings are then explored to cover new states, and to generate new seed interleavings.

MCR terminates when all seed interleavings have been explored and no new seed interleavings can be generated. For a new value constraint, there can be multiple interleavings in $\mathsf{MaxCausal}(\tau)$ that satisfy the constraint. To avoid generating redundant seed interleavings, MCR ensures that the prefix of each newly explored interleaving is always preserved and the generated seed interleaving is the shortest among all satisfiable interleavings.

To generate a seed interleaving from an input trace $\tau$, MCR consists of two main steps:

1. Encoding $\mathsf{MaxCausal}(\tau)$ and a new state constraint into a boolean formula;

2. Solving the formula by an SMT solver, and building a new seed interleaving from the solution.

Next, we describe the first step in Section 2.3 and illustrate the second step using an example in Section 2.4.

## 2.3 Constraint Encoding of MCR

For each event in the given trace $\tau$, MCR creates an order variable $O$ denoting its order in a certain feasible trace in $\mathsf{MaxCausal}(\tau)$. MCR then encodes $\mathsf{MaxCausal}(\tau)$ into a formula $\Phi^{mc}$ consisting of three types of first-order logical constraints over the order variables $O$: (1) must-happen-before constraints ($\Phi_{mhb}$); (2) lock-mutual-exclusion constraints ($\Phi_{lock}$); (3) data-validity constraints ($\Phi_{validity}$). $\Phi^{mc}$ is then conjoined with a new state constraint $\Phi_{state}$ to generate a final formula $\Phi$ that is used to produce a seed interleaving.

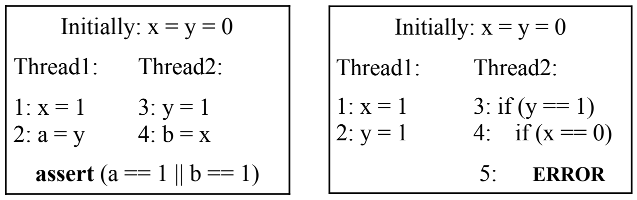
***Must-happen-before (MHB) constraints ($\Phi_{mhb}$).*** The $\Phi_{mhb}$ constraint ensures a minimal set of *happens-before* relations that events in any feasible interleaving must obey. It requires that (1) All events by the same thread should happen in the program order (obeying SC); (2) The *begin* event of a thread should happen after the *fork* event that starts the thread; (3) A *join* event for a thread should happen after the last event of the thread.

***Lock-mutual-exclusion constraints ($\Phi_{lock}$).*** The $\Phi_{lock}$ constraint ensures that events guarded by the same lock are mutually exclusive. It is constructed over the ordering of the *lock* and *unlock* events. More specifically, for each lock, MCR extracts all the *lock/unlock* pairs of events following the program order and constructs the following constraints for each two pairs $(l_1, u_1)$ and $(l_2, u_2)$: $O_{u_1} < O_{l_2} \vee O_{u_2} < O_{l_1}$.

***Data-validity constraints ($\Phi_{validity}$).*** The $\Phi_{validity}$ constraint ensures that all events in any trace in $\mathsf{MaxCausal}(\tau)$ are feasible. For an event $e$ to be feasible, all events that must-happen-before $e$ must be feasible, and every read event that $e$ depends on (excluding $e$ itself) should read the same value as it reads in $\tau$. Let $\prec_e$ denote the set of events that must-happen-before an event $e$, and consider a read event *r=read(t,x,v)* in $\prec_e$ on a memory address $x$ with value $v$ by thread $t$. Let $W^x$ denote the set of all writes to $x$, and $W^x_v$ the set of writes to $x$ with value $v$, the $\Phi_{validity}$ constraint for $e$ is encoded as $\bigwedge_{r \in \prec_e} \Phi_{value}(r, v)$, where $\Phi_{value}(r, v)$ is the state constraint that ensures $r$ to read a value $v$:

$$\Phi_{value}(r, v) \equiv \bigvee_{w \in W^x_v} (\Phi_{validity}(w) \wedge O_w < O_r$$
$$\bigwedge_{w \neq w' \in W^x} (O_{w'} < O_w \vee O_r < O_{w'}))$$

***New state constraints ($\Phi_{state}$).*** The key idea for MCR to eliminate redundant executions is enforcing at least one read event in each explored execution to read a new value so that no two executions reach the same state. MCR enumerates each read event in $\tau$ on the set of all values by the writes on the same memory address. For each value that is different from what it reads in $\tau$, a new state constraint is generated to ensure the read event to read the new value. Consider a read *r=read(t,x,v)* on $x$ with value $v$, and a value $v' \neq v$ written by any write on $x$, $\Phi_{state}$ is written as $\Phi_{value}(r, v')$. Since all such state constraints are generated, MCR ensures that no non-equivalent interleaving is missed. Hence, the entire state-space will be covered systematically by MCR.



```
Initially: x = y = 0
Thread1:     Thread2:
1: x = 1     3: y = 1
2: a = y     4: b = x
assert (a == 1 || b == 1)
```

**(a)** A TSO example

```
Initially: x = y = 0
Thread1:     Thread2:
1: x = 1     3: if (y == 1)
2: y = 1     4:   if (x == 0)
             5:     ERROR
```

**(b)** A PSO example

**Figure 3:** (a) shows a program with error under TSO, but correct under SC; (b) shows a program with error under PSO, but correct under SC and TSO.

## 2.4 Example

We use the example in Figure 3(a) to illustrate MCR. The program has 6 different executions (3 are redundant) under SC, but 24 different executions under TSO (20 are redundant). MCR is able to explore all the state-space under SC via only 3 executions, but it fails to expose the assertion violation that is only possible under TSO.

Let $e_i$ denote the event at the line number $i$. Given a trace $\tau = \langle e_1, \cdots, e_n \rangle$, MCR uses $n$ integer variables $\langle O_1, \cdots, O_n \rangle$ to denote the order in which the events happen in a certain execution. The value of $O_i$ represents the position of $e_i$ in a trace. If $O_i < O_j$, then $e_i$ will be executed before $e_j$ in the generated interleaving.

Suppose in the initial execution, MCR obtains the trace $\tau_0 = \langle e_1, e_2, e_3, e_4 \rangle$ under SC, and the program reaches the state ($a$=0,$b$=1). MCR constructs the MHB constraints $\Phi_{mhb} = O_1 < O_2 \wedge O_3 < O_4$. Since the trace contains two reads, $e_2$ ($R(y)$=0) and $e_4$ ($R(x)$=1), to generate new seed interleavings, MCR tries to enforce each of the two reads to read a different value in future executions. For example, for $e_2$, it adds the new state constraint $\Phi_{value} = O_3 < O_2$ to enforce $R(y)$ to read value 1 (written by $e_3$) instead of 0. By solving this constraint conjoined with $\Phi_{mhb}$, the SMT solver will return a solution such as $\{O_1 = 1, O_2 = 3, O_3 = 2\}$. From this solution, MCR will generate a new seed interleaving $e_1$-$e_3$-$e_2$ because $O_1 < O_3 < O_2$. By re-executing the program following this seed interleaving, MCR obtains a new trace $\tau_1 = \langle e_1, e_3, e_2, e_4 \rangle$ and reaches a new state ($a$=1,$b$=1). Then the exploration along this seed interleaving is finished because there is no new value that can be read by any read event in $\tau_1$. Similarly, the read event $e_4$ in $\tau_0$ generates a new seed interleaving $e_3$-$e_4$, which produces a new trace $\tau_2 = \langle e_3, e_4, e_1, e_2 \rangle$ that reaches a new state ($a$=1,$b$=0).

As we can see, through only three different executions, MCR successfully explores all three possible program states under SC: ($a$=0,$b$=1), ($a$=1,$b$=1) and ($a$=1,$b$=0). However, MCR misses the assertion violating state ($a$=0,$b$=0), which is feasible under TSO and PSO. To reach this state, there must be at least a reordering between $(e_1, e_2)$ or $(e_3, e_4)$. Neither of them is possible in the formulation of MCR because both of them violate the $\Phi_{mhb}$ constraint. Similarly, MCR cannot trigger the PSO assertion violation in Figure 3(b), because $e_1$ must-happen-before $e_2$ under SC. Next, we present the semantics of TSO and PSO in Section 3. We will show how our approach finds the errors under TSO and PSO in Section 4.

## 3. TSO and PSO

We present the operational semantics of TSO and PSO [23, 33] following the same spirit as previous work [4, 6]. We also discuss the relation of TSO and PSO to language memory models (*i.e.*, Java Memory Model) at the end of this section.

### 3.1 Hardware Memory Models

***Total Store Ordering (TSO).*** TSO allows a read to complete before an earlier write to a different memory location, but maintains a total order over writes and operations accessing the same memory location. There are four kinds of operations:

- **Store.** Whenever a thread $t_i$ executes a store operation, it does not update it to the shared main memory immediately. Instead, the store is buffered to the store buffer $B_i$ (which is a FIFO queue).
- **Load.** When a thread $t_i$ executes a load to a memory location $x$, it first checks its buffer $B_i$. If the buffer contains the store to $x$, then the load gets the latest value written to $x$ in the buffer; otherwise, the load obtains the value from the main memory.
- **Update.** An update operation flushes the store buffer into the main memory. It can happen at any point as long as the store buffer is not empty. The memory model allows any thread to non-deterministically perform the *update* operation any number of times at any state of the execution.
- **Fence.** Fences are special machine instructions that prevent reordering between the operations before and after the fence. A fence operation can only be executed when the buffer is empty.

Consider a concurrent program with $n$ threads $\boldsymbol{T} = t_1 \times t_2 \times \cdots \times t_n$ and each thread $t_i$ is associated with a store buffer $B_i$, forming a set of store buffers $\boldsymbol{B} = B_1 \times B_2 \times \cdots \times B_n$. Let $\mathbf{M} = M_1 \times M_2 \cdots \times M_k$ be the memory locations in the program, and each memory location can take value from a data domain. We define a system configuration as a tuple $\mathcal{C} = \langle T, M, B \rangle$, and the local configuration of thread $t_i$ as $\mathcal{C}_i = \langle M, B_i \rangle$ where $M$ is the current value in each memory location, and $B_i$ is the current value in the store buffer of thread $t_i$.

For two system configurations $\mathcal{C} = \langle T, M, B \rangle$ and $\mathcal{C}' = \langle T', M', B' \rangle$, we use the notation $\mathcal{C} \xrightarrow{op} \mathcal{C}'$ to denote the transition from $\mathcal{C}$ to $\mathcal{C}'$ by executing the operation *op*, where *op* is one of the four operations (store/load/update/fence) defined above by a certain thread. Consider that *op* is executed by thread $t_i$. The transition on the system configuration is the same as that on the local configuration of $t_i$: $\mathcal{C}_i \xrightarrow{op(t_i)} \mathcal{C}'_i$. We use $w(t_i, x, v)$/$r(t_i, x, v)$/$u(t_i, x, v)$/$fence(t_i)$ to denote these four operations respectively, meaning that thread $t_i$ writes/reads value $v$ to/from memory location $x$, updates the value $v$ to $x$ from the store buffer to the main memory, or performs the fence operation, respectively.

Let $B \oplus (x, v)$ denote buffering the write $(x, v)$ to the store buffer $B$, $B \ominus (x, v)$ flushing the write $(x, v)$ to the main memory from $B$, and $B = \varepsilon$ denote that $B$ is empty. Let $B(x)$ denote retrieving the value of the *most recent* buffered write to $x$ in $B$. Note that $B(x)$ can be *null* when there is no buffered write to $x$ in $B$. We use $\varnothing$ to denote the null value.

The operational model is defined as follows:

1. ***Store***: $\mathcal{C}_i \xrightarrow{w(t_i,x,v)} \mathcal{C}'_i$ *iff* $M' = M$ and $B'_i = B_i \oplus (x, v)$.
2. ***Load***: $\mathcal{C}_i \xrightarrow{r(t_i,x,v)} \mathcal{C}'_i$ *iff* $M' = M$, $B'_i = B_i$ and either one of the following two cases:
   (a) **Load from buffer**: $B_i(x) \neq \varnothing$ and $v = B_i(x)$.
   (b) **Load from memory**: $B_i(x) = \varnothing$ and $v = M[x]$.
3. ***Update***: $\mathcal{C}_i \xrightarrow{u(t_i,x,v)} \mathcal{C}'_i$ *iff* $B'_i = B_i \ominus (x, v)$ and $M' = M[x \hookleftarrow v]$.
4. ***Fence***: $\mathcal{C}_i \xrightarrow{fence(t_i)} \mathcal{C}'_i$ *iff* $B_i = \varepsilon$ and $M' = M$.

***Partial Store Ordering (PSO).*** PSO is similar to TSO except that it allows reordering writes on different memory locations. The operational model of PSO can be defined by slightly modifying the TSO model defined above. Under PSO, each thread has *multiple* store buffers, each of which corresponds to one unique memory location. In other words, each memory location is assigned with a store buffer. Two consecutive write operations on different memory locations can be buffered into different store buffers, allowing them to be executed out of the program order.

### 3.2 JMM on TSO/PSO Platforms

The motivation of our work stems from the real bug exhibited in Figure 1. Readers may be concerned that Java has its own memory model (JMM [28]) and the compiler and hardware reorderings should respect the JMM. However, hardware memory models are orthogonal to language memory models. For any language, as long as the compiler does not insert fences to prohibit reorderings, the hardware may exhibit TSO/PSO behaviors. Because the JMM allows the delayed stores as that in TSO and PSO [30, 34], the JVM inserts no barriers to disable the reorderings on TSO/PSO platforms. Consequently, the reordering can cause the bug in Figure 1 to occur. A fix to this bug is to declare both the fields $x$ and $y$ in Figure 1 as final. For final fields, the JVM inserts a barrier after the initialization of them. Thus, once an object is constructed, the values assigned to the final fields of the object are visible to the other threads. This prevents the bug from occurring in Figure 1.

## 4. OUR APPROACH

Our approach builds upon MCR but enables it to work for both TSO and PSO. There are two crucial differences between our approach and the original MCR [21]:

1. We relax the *must-happens-before* (MHB) relation between events to capture the semantics of TSO and PSO when producing the seed interleavings.
2. We develop novel replay algorithms for TSO and PSO interleavings that allow the reordering of events by the same thread.

In this section, we first describe how to relax the MHB constraints to allow the semantics of TSO and PSO defined in Section 3. We then present our replay algorithms. Finally, we discuss the limitation of our approach.

### 4.1 Relaxation of the MHB Constraints

To encode the semantics of TSO and PSO, we relax the MHB constraints $\Phi_{mhb}$ of MCR (recall Section 2.3). Specifically, we decompose $\Phi_{mhb}$ into two components:

$$\Phi_{mhb} = \Phi_{mem} \land \Phi_{sync}$$

where (1) the memory operation constraint ($\Phi_{mem}$) captures the reordering semantics allowed by different memory models (TSO or PSO); (2) the synchronization constraint ($\Phi_{sync}$) captures the *happens-before* relation entailed by synchronizations. $\Phi_{sync}$ is common for all memory models (e.g., SC/TSO/PSO).

***Constraints on memory operations ($\Phi_{mem}$).*** Under TSO, following the operational semantics defined in Section 3, we construct $\Phi_{mem}$ with four rules: (1) *write-to-write constraints* ($\Phi_{ww}$). For all writes by the same thread, their order should be consistent with the program order. (2) *memory location constraints* ($\Phi_{addr}$). For all the reads and writes by the same thread that access the same memory address, they should follow the program order. (3) *read-to-read constraints* ($\Phi_{rr}$). All read operations from the same thread should follow the program order. (4) *read-to-write constraints* ($\Phi_{rw}$). Any read operation and its following write operation from the same thread should follow the program order. Together, $\Phi_{mem}$ is represented as the conjunction of these four constraints:

$$\Phi_{mem} = \Phi_{ww} \land \Phi_{rr} \land \Phi_{rw} \land \Phi_{addr}$$

PSO is a further relaxation of TSO. PSO not only allows the write-to-read reordering allowed by TSO, but also the

| | | |
|---|---|---|
| **MCR** | | $O_1<O_2, O_3<O_4$ |
| | **SC** | $O_1<O_2, O_3<O_4$ |
| **Our Approach** | **TSO** | $O_1, O_2, O_3, O_4$ |
| | **PSO** | $O_1, O_2, O_3, O_4$ |

**(a)**

| | | |
|---|---|---|
| **MCR** | | $O_1<O_2, O_3<O_4$ |
| | **SC** | $O_1<O_2, O_3<O_4$ |
| **Our Approach** | **TSO** | $O_1<O_2, O_3<O_4$ |
| | **PSO** | $O_1, O_2, O_3<O_4$ |

**(b)**

**Figure 4:** The *must-happen-before* constraints constructed by MCR and our approach on the TSO and PSO examples in Figure 3(a) and Figure 3(b), respectively.

reordering of write-to-write to different memory locations. Therefore the only difference between the $\Phi_{mem}$ constraint under TSO and PSO is the rule $\Phi_{ww}$. In PSO, $\Phi_{ww}$ ensures only that all writes to the same memory location from the same thread should follow the program order.

***Constraints on synchronizations ($\Phi_{sync}$).*** For all synchronizations (*i.e.*, *lock*/*unlock* and *begin*/*end*) by the same thread, they should always be executed in the program order. Moreover, for each synchronization, all its proceding reads and writes should always happen before it, and all its following reads and writes should always happen after it.

### 4.2 New States under Relaxed MHB

To show the difference brought by the relaxed MHB constraints compared to the original MCR, we use the example in Figure 3(a) again to illustrate how it enables exposing the TSO and PSO errors which MCR fails to expose. Same as in Section 2.4, let us assume that $\tau_0 = \langle e_1, e_2, e_3, e_4\rangle$ is observed as the initial trace. Figure 4 shows a comparison between the MHB constraints on $\tau_0$ constructed by MCR and by our approach for TSO and PSO, respectively. Consider the read $e_4$ ($R(x)$=1). To make $R(x)$=0 in the new seed interleaving, MCR enforces $e_4$ to happen before $e_1$ by the constraint $O_4 < O_1$. Under TSO, because $e_3$ does not necessarily happen before $e_4$, our approach does not enforce $O_3 < O_4$ (as shown in Figure 4a) compared to MCR. As a result, the generated new seed interleaving by our approach is just $e_4$, whereas it is $e_3$-$e_4$ by MCR. By replaying the program with the new seed interleaving $e_4$, our approach will explore a new execution and generate a new trace $\tau_1$ starting with $e_4$, such as $\tau_1 = \langle e_4, e_1, e_2, e_3\rangle$. In this case, $\tau_1$ reaches the state ($a$=0,$b$=0), which violates the TSO assertion.

Likewise, under PSO, to expose the error in Figure 3b, our approach generates an execution $\tau_1 = \langle e_2, e_3, e_4, e_1\rangle$ because of the reordering between $e_1$ and $e_2$ under PSO.

### 4.3 Deterministic Replay

A key challenge in extending MCR from SC to TSO and PSO lies in how to replay the TSO and PSO interleavings. Under the original MCR, the interleaving is abstracted as a sequence of schedule choices, with each choice representing a thread ID by the corresponding operation on a shared variable. Before a thread executes an operation on a shared location, it is blocked first, and then the scheduler queries the seed interleaving to decide which thread to execute next. For SC, since the global order in the generated seed interleaving is consistent with the program order, this replay strategy guarantees that the operation chosen by the scheduler exactly matches with the event in the interleaving. However, under TSO/PSO, because operations can be executed out of the program order, this simple global-ordering based replay approach no longer works. To realize the reordering, we rely on a store buffer (a FIFO queue) assigned to each thread to delay the execution of a store. The difficulty comes from the non-determinism of the *update* operation (recall Section 3) because it can happen any time at any point of the execution to flush the store buffer. To deterministically enforce a seed TSO/PSO interleaving, there are two key issues to be addressed: (1) *when to buffer a write*; and (2) *when to flush the buffered write to the main memory*.

To solve this problem, we first extend the original abstraction of the SC interleaving in MCR by adding the memory location information – *addr* – to each operation. The new abstraction of *interleaving* is defined as follows:

**Definition 1.** *An interleaving is a sequence of schedule choices, with each schedule choice $c(tid, addr)$ consisting of a thread ID, $tid$, and a memory location, $addr$, that is expected to be accessed by the corresponding operation.*

The key idea of our TSO and PSO replay algorithms is to use the accessed memory location to decide whether to buffer a write by checking it against the information in the seed interleaving.

***Store Buffering/Updating.*** Before performing a store operation, we first check if the memory location accessed by this operation is the same as the one in the seed interleaving. If yes, we can flush the store to the main memory. Otherwise, we buffer the store in the store buffer (a FIFO Queue). At this point, we do not update the schedule choice since the operation has not been executed from the view of the interleaving. Later, when the address by the event in the interleaving matches with the one buffered in the FIFO queue, we flush the value to the main memory and also update the schedule choice to the next one.

***Fence.*** The operational model of TSO and PSO requires that before performing a fence operation, all the buffered stores in the *store buffer* should be flushed into the memory. However, in our approach, the re-execution of a program is controlled by a given interleaving, and our replay algorithm guarantees that when the scheduler meets a fence operation, the buffer must be empty. The reason is that all events that occur before a fence should happen before the fence and we have already constrained all such events to happen before the fence in the formula (recall Section 4.1). Therefore, when the scheduler is about to execute a fence of an interleaving, all the events before this fence have already been executed

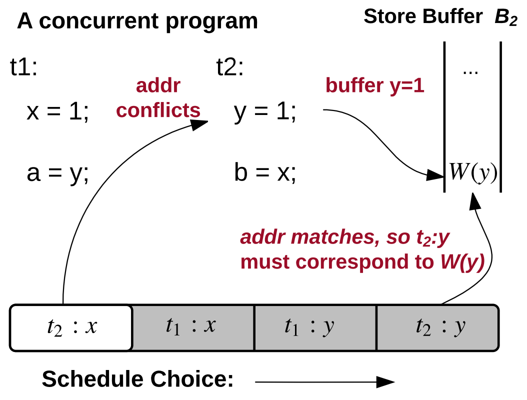


**Figure 5:** An example to illustrate Theorem 1 and Theorem 2. $B_2$ is the store buffer associated with thread $t_2$.

(all buffered writes have already been flushed), and thus the buffer is empty at the moment.

Based on the new abstraction above, we can prove the following two theorems to guide our replay algorithms and to guarantee their correctness. Theorem 1 guides our algorithm to buffer writes and Theorem 2 guides our algorithm to flush the buffered writes to memory.

**Theorem 1.** *At replay, when the program counter (PC) points to an event, say $e_i$, corresponding to the choice of the schedule, say $c_j$, if $addr(e_i) \neq addr(c_j)$, then $e_i$ must be a write operation and it needs to be buffered.*

*Proof:* If $e_i$ is a read or a fence operation, it implies that a later operation $c_j$ (later according to the program order) is allowed to happen before a read/fence operation. This contradicts with the TSO and PSO operational models defined in Section 3. Hence $e_i$ must be a write, and a certain operation matching with $c_j$ that accesses a different location should be executed before $e_i$. Therefore, $e_i$ must be buffered.

**Theorem 2.** *When considering a schedule choice $c_j$ in the seed interleaving, if $addr(c_j)$ equals to the memory location of the write at the head of the store buffer, then $c_j$ must correspond to that buffered write.*

*Proof:* By contradiction. Suppose that $c_j$ is a read or a fence. Since there is a write $w$ in the store buffer by the same thread that accesses the same address as $c_j$ does, it means that the read (or fence) is allowed to happen before the write $w$ which is before it. This again contradicts with the operational models of TSO and PSO. Therefore, $c_j$ must be a write. Similarly, $c_j$ cannot correspond to any other write in the store buffer, otherwise $c_j$ would be allowed to be executed before its preceding writes. Hence, $c_j$ must correspond to the buffered write $w$ and $w$ should be flushed.

**Example.** Figure 5 illustrates the two theorems above. When considering the schedule choice $t_2 : x$, the instruction $y = 1$ is to be executed. However, their addresses do not match, which implies that the write $y = 1$ should be buffered

(Theorem 1). When considering the last schedule choice $t_2 : y$, its address matches with that of the write at the head of thread $t_2$'s store buffer $B_2$, which implies that $t_2 : y$ is a buffered write and has to be flushed (Theorem 2).

## 4.4 TSO Replay

To replay TSO interleavings, we associate each thread $t_i$ with a FIFO queue (to simulate the store buffer $B_i$) and we assume the queue is unbounded. The *interleaving* here is a seed interleaving generated based on the solution given by the SMT solver. Each interleaving (recall Definition 1) consists of a sequence of schedule choices $c(tid, addr)$. The program is executed under the control of an application-level scheduler to enforce the schedule choices specified in the interleaving.

**Algorithm 1** TSO replay algorithm

```
Input:  A seed interleaving $S$ – schedule choices
Output:  a new trace by logging the instruction executed
 1: Initial: $index = 0$    //global
 2: while ($index < S.length$) do
 3:     $c \leftarrow S[index]$
 4:     $Inst \leftarrow PC(c)$    //guided by the schedule choice
 5:     $i = tid(Inst)$
 6:     $x = addr(Inst)$
 7:     if $Inst$ is a store then
 8:         $v = value(Inst)$  //the value of the store
 9:         if $addr(Inst) == addr(c)$ then
10:             $Write(x, v)$
11:             $index = index+1$
12:             $updateCheck\_TSO(S)$
13:         else    //buffer the store
14:             $B_i \leftarrow B_i \oplus (x, v)$
15:     else if $Inst$ is a load then
16:         if $x$ in $B_i$ then
17:             $v = B_i(x)$    //read the most recent value from buffer
18:         else
19:             $v = mem(x)$    //read the value from memory
20:         $index = index+1$
21:         $updateCheck\_TSO(S)$
22:     else //fence – the buffer should be empty
23:         $Fence()$
24:         $index = index+1$
```

Algorithm 1 shows how we replay a TSO interleaving. The key idea is to determine whether to buffer or to update a *store* by comparing its memory location with that specified in the schedule choice. We use a variable *index* to indicate the current position of the interleaving. The *index* is initialized to 0 and incremented by one each time when an instruction is executed (except for the store buffer operation). Before the program executes a load or a store on a shared variable, the program is blocked and the schedule choice $c$ given by the interleaving is queried to decide the next instruction. The next instruction is chosen by the program counter via the schedule choice (see the statement $PC(c)$ at line 4). Before executing an instruction, the algorithm proceeds depending on its type. If the instruction is a *store*, the algorithm first checks whether the memory location of this store and that specified in the schedule choice are equal or not. If they are equal, we write the value to the memory (line 10). Otherwise we buffer the store into the thread's FIFO queue $B_i$. If the instruction is a *load*, the algorithm first checks if there is a buffered write in $B_i$ that writes to the same address. If yes, the most recent buffered value is returned; otherwise, the value from the main memory is returned. For *fence* instructions, we simply proceed without the need to flush the buffer because as discussed in Section 4.3 the buffer must be empty.

**Algorithm 2** After a load/store, check whether there are pending stores in the buffer that need to be updated.

```
 1: function UPDATECHECK_TSO($S$)
 2:     $c \leftarrow S[index]$  //return if index out of bound
 3:     $i = tid(c)$
 4:     while ($addr(c) == addr(B_i[0])$) do
 5:         $B_i \leftarrow B_i \ominus (x, v)$
 6:         $flush(x, v)$  //flush to memory
 7:         $index = index+1$
 8:         $c \leftarrow S[index]$  //return if index out of bound
 9:         $i = tid(c)$
10: function UPDATECHECK_PSO($S$)
11:     $c \leftarrow S[index]$  //return if index out of bound
12:     $i = tid(c)$
13:     $j = varId(c)$
14:     while ($addr(c) == addr(B_i^j[0])$) do
15:         $B_i^j \leftarrow B_i^j \ominus (x, v)$
16:         $flush(x, v)$  //flush to memory
17:         $index = index + 1$
18:         $c \leftarrow S[index]$  //return if index out of bound
19:         $i = tid(c)$
20:         $j = varId(c)$
```

Each time afte a *load* or *store* is executed, our algorithm checks if there are stores in the FIFO queue that should be flushed. The function $UpdateCheck\_TSO$ in Algorithm 2 shows the process for flushing the buffered stores for TSO and PSO. Recall Theorem 2 that when the current schedule choice has the same memory location as that of the store at the head of the buffer, the expected operation must be a store that has been buffered. We hence follow this condition to detect all such stores and update them to the memory.

***Termination.*** Note that our algorithm just replays the instructions within the interleaving, it terminates when $index = S.length$. For those outside of the interleaving, they are executed following the program order. The number of while-loop iterations (line 2) is determined by the $index$, which specifies the schedule choice. Although the $index$ is unchanged when buffering a store (line 13), it will eventually be increased to the size of the schedule, which terminates the algorithm. Each time after we execute or buffer an instruction, the program counter will be updated to point to the next instruction controlled by the schedule choice (line 4). Although the $index$ is not changed, the address or the type (read/write) of the operation will change, which leads to the execution of a read/write, and eventually the execution of the update function which increases the index.

The correctness of our algorithm is guaranteed by the following theorem.

**Theorem 3.** *Algorithm 1 correctly replays all the events in the given TSO interleaving.*

*Proof:* Theorem 1 and Theorem 2 guarantee that all events in the given interleaving are replayed in the order as specified in the interleaving. To prove Theorem 3, we only need to prove that all the events in the interleaving are replayed, *i.e.*, no event is missed. In our replay algorithm, each time after an event is executed, the $updateCheck\_TSO(S)$ subroutine updates (*i.e.*, execute) the buffered events until the index points to an event that is not buffered. Suppose there exists an event $e$ that is not executed when the replay algorithm is finished. If $e$ corresponds to a non-buffered event, $e$ should be executed directly when it is chosen by the $index$. On the other hand, if $e$ corresponds to a buffered store, there must exist a nearest non-buffered event $e'$ preceding $e$ in the interleaving. After $e'$ is executed, $updateCheck\_TSO(S)$ subroutine will execute $e$. Thus, in any case, no event will be missed.

**Example.** To illustrate the algorithm, consider a TSO interleaving of the program in Figure 3a: $e_4, e_1, e_2, e_3$, which corresponds to the sequence of schedule choices: ($t_2$,$x$),($t_1$,$x$),($t_1$,$y$),($t_2$,$y$). When replaying this interleaving, the schedule choice ($t_2$,$x$) guides the program to execute the instruction $y = 1$ at line 3. Since the addresses do not match, $y = 1$ is buffered and the schedule index does not change. When the program reaches the instruction $b = x$ at line 4, since it is a load operation, $b = x$ is executed directly. Similarly, $x = 1$ and $a = y$ at lines 1 and 2 are executed under the schedule choices ($t_1$,$x$),($t_1$,$y$). After $a = y$ is executed, the algorithm detects that the schedule choice ($t_2$,$y$) corresponds to the buffered write $y = 1$ in the FIFO queue. Therefore, $y = 1$ is updated to the memory.

## 4.5 PSO Replay

Replaying PSO interleavings is similar to that for TSO. The only difference is that under PSO, each thread is associated with multiple FIFO queues with each queue corresponding to one unique memory location. For a thread $t_i$ accessing a memory locations $m_k$, we assign a FIFO queue $B_i^k$ for the memory location $m_k$. Algorithm 3 shows the replay process. The key difference from Algorithm 1 is that when buffering a store under PSO, the algorithm needs to buffer the store to the FIFO queue corresponding to the memory location accessed by the store. The function $UpdateCheck\_PSO$ in Algorithm 2 shows the process for flushing the buffered stores under PSO (similar to the process for TSO). We use $varId()$ to get the unique ID assigned to each variable by each thread, and the buffer $B_i^j$ corresponds to a variable belonging to thread $i$ with ID $j$.

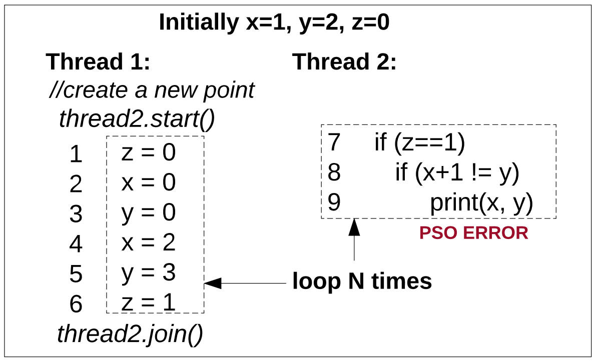
**Algorithm 3** PSO replay algorithm

```
Input: A seed interleaving $S$ – schedule choices
Output: a new trace by logging the instruction executed
1: Initial: $index = 0$    //global
2: while ($index < S.length$) do
3:     $c \leftarrow S[index]$
4:     $Inst \leftarrow PC(c)$    //guided by the schedule choice
5:     $x_j = addr(Inst)$
6:     $i = tid(Inst)$
7:     $j = varId(x_j)$
8:     if $Inst$ is a store then
9:         $v = value(Inst)$  //the value of the store
10:        if $addr(Inst) == addr(c)$ then
11:            $Write(x_j, v)$
12:            $index = index + 1$
13:            $updateCheck\_PSO(S)$
14:        else //buffer the store
15:            $B_i^j \leftarrow B_i^j \oplus (x_j, v)$
16:    else if $Inst$ is a load then
17:        if $x_j$ in $B_i^j$ then
18:            $v = B_i^j(x_j)$    //read the most recent value from buffer
19:        else
20:            $v = mem(x_j)$    //read the value from memory
21:        $index = index + 1$
22:        $updateCheck\_PSO(S,index)$
23:    else //fence – the buffer should be empty
24:        $Fence()$
25:        $index = index+1$
```

## 4.6 Discussion

We note that our approach is not optimal for minimizing the redundancy under TSO and PSO, albeit MCR is optimal for SC. The root problem is that under TSO and PSO, the generated seed interleavings are shorter than that under SC, which results in the possibility that two distinct seed interleavings may reach the same state. Consider again the example in Figure 3(a). In the initial execution $\tau_0 = \langle e_1, e_2, e_3, e_4 \rangle$, there exists two reads $e_2$ ($R(y)$=0) and $e_4$ ($R(x)$=1). If we force the read on $y$ ($e_2$) to read value 1 which requires $e_3$ to happen before $e_2$, our approach will generate a seed interleaving $e_3$-$e_2$. Likewise, if we force the read on $x$ ($e_4$) to read value 0, which requires $e_4$ to happen before $e_1$, our approach will generate a seed interleaving $e_4$.

If we continue with the seed interleaving $e_3$-$e_2$, we will generate two more executions:



```
Initially x=1, y=2, z=0
Thread 1:                    Thread 2:
//create a new point
thread2.start()
1  z = 0                     7  if (z==1)
2  x = 0                     8     if (x+1 != y)
3  y = 0                     9        print(x, y)
4  x = 2                              PSO ERROR
5  y = 3        <- loop N times
6  z = 1
thread2.join()
```

**Figure 6:** A simplified version of the program in Figure 1. An execution 1-2-6-7-8-3-4-5-8-9 can trigger this error under PSO.

- $\tau_1 = \langle e_3, e_2, e_1, e_4 \rangle$ ($a$=1,$b$=1);
- $\tau_2 = \langle e_3, e_2, e_4, e_1 \rangle$ ($a$=1,$b$=0).

And if we continue with the seed interleaving $e_4$, we will generate another two executions:

- $\tau_3 = \langle e_4, e_1, e_2, e_3 \rangle$ ($a$=0,$b$=0);
- $\tau_4 = \langle e_4, e_3, e_2, e_1 \rangle$ ($a$=1,$b$=0).

As we can see, under TSO, our approach explores five executions to cover the whole state-space. However, the optimal solution should only explore four executions, because there are only four unique states. In our approach, $\tau_2$ and $\tau_4$ are equivalent to each other, both of which reach the state ($a$=1,$b$=0).

The only difference between these two redundant executions is the permutation of the two seed interleavings: $e_3$-$e_2$ and $e_4$, where $e_3$-$e_2$ targets the value read from $y$ and $e_4$ the value read from $x$. Since these two seed interleavings are non-overlapping and are permutable, they lead to the same state. However, it is difficult to prune this type of redundancy in the current MCR, because the seed interleavings are generated independently without considering their permutations. A potential way to eliminate this redundancy for TSO and PSO would be to merge multiple independent seed interleavings into a single one. Nevertheless, this type of redudancy only accounts for a minor portion of the explored executions, because the space of seed interleavings is significantly smaller than the whole interleaving space. As we will show in our experiments in Section 6, even with this redundancy, MCR under TSO and PSO is much more effective than existing approaches on both popular benchmarks and real programs.

## 5. Case Study

In this section, we present a case study of our approach on the real PSO bug in Figure 1. We also compare our approach with the DPOR algorithm for relaxed memory models by Zhang et al. [38]. We show that our approach is much more effective

| | |
|---|---|
| $\Phi_{mhb}$(SC, TSO) | $O_1<O_2<O_3<O_4<O_5<O_6$ <br> $O_7<O_8^1<O_8^2$ |
| $\Phi_{mhb}$(PSO) | $O_1<O_6$ $O_2<O_4$ $O_3<O_5$ <br> $O_7<O_8^1<O_8^2$ |
| $\Phi_{validity}$ | $O_6<O_7$ |
| $\Phi_{state}$ | $O_2<O_8^1 \wedge (O_4<O_2 \vee O_8^1<O_4)$ |

$$\Phi = \Phi_{mhb} \wedge \Phi_{validity} \wedge \Phi_{state}$$

**Figure 7:** The generated constraints by our approach under three different memory models for the example in Figure 6.

than the DPOR algorithm for both state-space exploration and bug finding.

To make the problem more clear, we simplify the program to its equivalent form, as shown in Figure 6. Note that the simplified example in Figure 6 is slightly different from the program in Figure 1, but it exactly presents how the PSO bug occurs in the original program. Lines 2 and 3 of the example in Figure 6 simulate the instructions for constructing a new *Point* object. Lines 4 and 5 write the initial values to the fields of the object. We use an integer variable $z$ to indicate whether or not the object is constructed. If $z = 1$, it means that the object is constructed. In our case, we do not simulate the *while* statements but just update the values of $x$ and $y$ once, which is enough to reveal the bug. We set the loop iterations to $N$ to control the complexity of the program.

Suppose that in the initial execution the two threads run sequentially following the program order, we obtain a trace as below[3]:

$$\tau_0 = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8^1, e_8^2 \rangle$$

where $e_i$ corresponds to an event performed at line $i$, and $e_8^1$ and $e_8^2$ correspond to the first and second (read to $x$ and $y$) events at line 8, respectively. In the initial trace, there are three reads: $e_7$ ($R_7(z)$=1), $e_8^1$ ($R_8(x)$=2) and $e_8^2$ ($R_8(y)$=3). The index of $R$ corresponds to the line number of the statement.

Consider the read $e_8^1$ ($R_8(x)$=2). To generate a new seed interleaving, we first try to enforce $e_8^1$ to read a different value 0 (by $e_2$ which writes 0 to $x$). Figure 7 shows the corresponding constraints constructed by MCR with our approach for the three different memory models (SC, TSO and PSO). $\Phi_{state}$ ensures that $R_8^1(x)$ reads 0 by enforcing $e_2$ to happen before $e_8^1$ and $e_4$ to either happen before $e_2$ or after $e_8^1$. $\Phi_{validity}$ ensures that $R_7(z)$ reads the same value as it reads in the trace $\tau_0$.

For SC and TSO, the generated constraint formula is not satisfiable. However, for PSO, the SMT solver returns a solution $\{O_1 = 0, O_2 = 0, O_6 = 1, O_7 = 2, O_8^1 = 3\}$. Based on this solution, we generate a new seed interleaving to continue with: $e_1$-$e_2$-$e_6$-$e_7$-$e_8^1$.

[3] We set $N$ to 1 in this case to simplify the presentation.

**Table 1:** Experimental results between our approach and DPOR on the program in Figure 6 with N from 1 to 4. The numbers indicate the number of executions explored by each approach. The symbols indicate timeout in one hour ($*$), found the PSO bug (✔) or not (✘), and threw an exception ($\ominus$).

| Loop times | DPOR | | | | MCR (our approach) | | | |
|---|---|---|---|---|---|---|---|---|
| | SC | TSO | PSO | Error found? | SC | TSO | PSO | Error found? |
| N=1 | 4 | 4 | $\ominus$ | ✘ | 2 | 2 | 10 | ✔ |
| N=2 | 105 | 105 | $\ominus$ | ✘ | 43 | 43 | 89 | ✔ |
| N=3 | 4282 | 4282 | $\ominus$ | ✘ | 296 | 296 | 819 | ✔ |
| N=4 | 14840* | 14840* | $\ominus$ | ✘ | 2767 | 2767 | 8420 | ✔ |

By re-executing the program with this seed interleaving, we obtain a new trace:

$$\tau_1 = \langle e_1, e_2, e_6, e_7, e_8^1, \mathbf{e_3}, \mathbf{e_4}, \mathbf{e_5}, \mathbf{e_8^2}, \mathbf{e_9^1}, \mathbf{e_9^2}\rangle$$

The events after the seed interleaving are newly explored events. Among these new events there are again three reads: $e_8^2$ ($R_8(y)$=3), $e_9^1$ ($R_9(x)$=2) and $e_9^2$ ($R_9(y)$=3). Since $R_8(x) = 0$ and $R_8(y) = 3$ at line 8, the *if* condition is satisfied and hence the error is triggered. However, we note that this bug is quite elusive. Before the read $R_9(x)$ at line 9, the buffered write to $x$ has already been flushed to the memory. When the program executes line 9, we have $R_9(x) = 2$ and $R_9(y) = 3$, which contradicts with the *if* condition.

Table 1 reports the results comparing our approach with the DPOR algorithm on the number of explored executions and on whether the approach can trigger the error or not under PSO. We set the number of loop iterations from 1 to 4. Our results show that as the number of the loop iterations increases, the number of executions explored by both of the two approaches increases dramatically (2 to 8420 for MCR and 4 to more than 14840 for DPOR). The reason is that the state-space of the program significantly increases as more reads and writes are executed. However, for all three memory models, MCR can finish exploring the state-space in a few seconds, whereas when $N = 4$, DPOR fails to finish the exploration in an hour after exploring 14840 executions under SC and TSO, and under PSO it terminates early by throwing an exception (likely due to an implementation bug in the *rInsepct* tool [38]). Moreover, our approach takes only three executions to trigger the PSO error, whereas DPOR fails to find the error by throwing an exception.

## 6. EXPERIMENTS

We have implemented our approach based on the original MCR [21] for multithreaded Java programs with ASM [1] for dynamic bytecode instrumentation and Z3 [14] for constraint solving. We extended MCR from SC to TSO and PSO by relaxing the *must-happen-before* constraints and implementing the TSO and PSO replay algorithms presented in Section 4. We have evaluated our approach on a collection of popular benchmarks and real applications shown in Table 2. *Dekker*, *Lamport*, *Bakery* and *Peterson* are four classic solutions to mutual exclusion problems from the previous work [4, 11, 38], all of which are intensively racy programs. *StackUnsafe* contains improper stack operations collected from [38]. *RVExample* is the motivating example in the original MCR paper [21]. *Example* is the real PSO bug example in Figure 1. The other six benchmarks are real programs used in previous concurrency studies [21, 22], including a large application – *Weblech*.

In the rest of this section, we first describe our evaluation methodology and then report our experimental results.

### 6.1 Evaluation Methodology

Our evaluation aims to answer the following three research questions:

1. How effective is our approach for exploring the state-space of concurrent programs?
2. How effective is our approach for finding TSO and PSO errors?
3. How scalable is our approach for real programs?

For the first question, we compared our approach with the most recent development of DPOR by Zhang et al. [38], which extends the original DPOR algorithm [16] with sleep-set reduction for TSO and PSO. Because their *rInspect* tool is implemented for C/C++, we carefully transformed seven standard benchmarks from Java to C/C++ or reversely for the comparison.

For the second question, we compared the number of executions required by different approaches to expose the injected or known errors in each benchmark. We injected assertion violations in the critical sections of four mutual exclusion programs for different memory models. For those benchmarks with known errors (e.g., *StackUnsafe* and *RVExample*), we directly used those errors for the evaluation. Besides DPOR, we also compared our approach with SATCheck [15], a recent SAT-based stateless model checking approach. SATCheck is a branch-driven approach that aims to cover all branches and all the unknown behaviors of the uninterpreted functions by systematically exploring thread schedules under SC and TSO. However, we found that the SATCheck tool missed executions during testing, especially when the benchmarks become more complicated, e.g., when the program has more conditional paths. Also, since SATCheck runs on C/C++ programs that use primitive

**Table 2:** Benchmarks

| Program | LoC | #Thrd | #Evt | Description |
|---|---|---|---|---|
| Dekker | 119 | 3 | 56 | Two critical sections with 3 shared variables. |
| Lamport | 162 | 3 | 40 | Two critical sections with 4 variables. |
| bakery | 119 | 3 | 27 | n critical sections using 2n shared variables. We take n=2. |
| Peterson | 94 | 3 | 72 | Two critical sections with 3 variables |
| StackUnsafe | 135 | 3 | 34 | Unsafe operations on a stack by two threads, which cause the stack underflow. |
| RVExample | 79 | 3 | 32 | An example from original MCR [21], which contains a very tricky error |
| Example | 73 | 2 | 44 | The example program from Figure 6 with loop number from 1 to 4. |
| Account | 373 | 5 | 51 | Concurrent account deposits and withdrawals suffering from atomicity violations. |
| Airline | 136 | 6 | 67 | A race condition causing the tickets oversold. |
| Allocation | 348 | 3 | 125 | An atomicity violation causing the same block allocated or freed twice. |
| PingPong | 388 | 6 | 44 | The player is set to null by one thread and dereferenced by another throwing NPE. |
| StringBuf | 1339 | 3 | 70 | An atomicity violation in Java StringBuffer causing StringIndexOutOfBoundsException. |
| Weblech | 35K | 3 | 2045 | A tool for downloading websites and enumerating standard web-browser behavior. |

reads/writes to access the shared memory, it needs sophisticated instrumentations to identify operations on shared variables, which is done manually in SATCheck. For comparison, we carefully transformed the seven benchmarks to the input format of SATCheck.

For the third question, we tested our approach on six real programs. We evaluated our approach on these programs under TSO and PSO, in addition to SC which is supported in the original MCR. Because none of the other two tools can support complex real applications and both of them work for C/C++ programs, we were not able to compare our results with the other approaches.

All experiments were conducted on a MacBook with 2.6 GHz Intel Core i5 processor, 8 GB DDR3 memory and JDK 1.7. All results were averaged over three runs.

## 6.2 Results of State Space Exploration

Table 3 summarizes the results of state-space exploration for the first seven benchmarks in Table 2. The first three columns report the results of DPOR, the three columns in the middle report the results of our approach, and the last three columns report the comparison between the two approaches. On average, MCR requires 5X to 10X (as much as 30X) fewer executions than DPOR to explore the entire state-space. For *RVExample*, which contains a very tricky error with loops, DPOR takes almost 2,000 executions, while MCR only takes 57 executions under SC. Moreover, the *rInspect* tool cannot finish under TSO and PSO by throwing a socket exception. For our *Example* in Figure 1, MCR takes 2,767 executions under SC and TSO, and 8,420 executions under PSO. Because the tools are implemented in different languages, it is difficult to compare the runtime speed between them. We hence focused on evaluating the effectiveness of our approach in reducing the number of executions but not the runtime performance. In the original MCR paper [21], it has shown that MCR outperforms DPOR in terms of runtime speed.

## 6.3 Results of Bug Finding

Table 4 summarizes the results of bug finding for the first seven benchmarks in Table 2. The data in the table presents the number of executions taken to find the bug. Overall, our approach takes much fewer executions than the other two approaches to find the errors. Moreover, our technique is able to find all the known errors and injected assertion violations, whereas DPOR fails to find the errors in *Example* and *RVExample* by throwing exceptions, and SATCheck fails to find those errors by either throwing segmentation faults or repeating the same execution forever. For example, for *Dekker*, *Peterson* and *RVExample*, the SATCheck tool gets stuck with the same execution and runs forever. Due to the implementation problem of the SATCheck tool and complexity of the transformation of the benchmarks, at the moment, it is difficult to make direct and fair comparisons between SATCheck and our technique. For *RVExample*, DPOR takes 301 executions to find that tricky error, while our approach takes only 53 executions.

## 6.4 Results on Real Programs

Table 5 reports the number of executions taken by MCR to explore the state-space of the six real programs under SC, TSO and PSO as well as the number of data races found during the exploration. MCR stops exploration when all state-space of the program has been explored or it triggers a bug in the program that leads to a runtime exception. Overall, MCR scales well to these real programs, and it is highly effective in exploring the state-space and finding bugs including data races in these programs. For example, for *Account*, MCR took only 7 executions to explore the whole state-space under SC, and 9, 11 under TSO and PSO, and found 3 data races. For *Weblech*, which contains over 2K critical events, MCR finished after explorating 185, 106 and 113 executions, respectively, under SC, TSO and PSO, and found 6 data races. The reason that MCR explored fewer executions under TSO and PSO than that under SC is that bugs in *Weblech* that

**Table 3:** Results of state-space exploration between our approach and DPOR. $*$ means timeout in one hour and ⊖ an exception happened before finishing the experiment.

| **Program** | **DPOR** | | | **MCR (our approach)** | | | **SpeedUp** | | |
|---|---|---|---|---|---|---|---|---|---|
| | SC | TSO | PSO | SC | TSO | PSO | SC | TSO | PSO |
| **Dekker** | 248 | 252 | 508 | 62 | 98 | 155 | 4.0X | 2.6X | 3.3X |
| **Lamport** | 128 | 208 | 2672 | 14 | 91 | 102 | 9.1X | 2.3X | 29.4X |
| **Bakery** | 350 | 1164 | 2040 | 77 | 158 | 165 | 4.5X | 7.1X | 12.4X |
| **Peterson** | 36 | 95 | 120 | 13 | 18 | 19 | 2.8X | 5.3X | 6.3X |
| **StackUnsafe** | 252 | 252 | 252 | 29 | 46 | 108 | 8.7X | 5.5X | 2.3X |
| **RVExample** | 1959 | ⊖ | ⊖ | 57 | 64 | 70 | 34.4X | ⊖ | ⊖ |
| | 4 | 4 | ⊖ | 2 | 2 | 10 | 2.0X | 2.0X | ⊖ |
| **Example** | 105 | 105 | ⊖ | 43 | 43 | 89 | 2.4X | 2.4X | ⊖ |
| (N = 1 to 4) | 4282 | 4282 | ⊖ | 296 | 296 | 819 | 14.5X | 14.5X | ⊖ |
| | 14840$^*$ | 14840$^*$ | ⊖ | 2767 | 2767 | 8420 | 5.4X | 5.4X | ⊖ |
| **Avg**. | 435 | 394 | 1118 | 42 | 79 | 103 | 10.4X | 5.0X | 10.9X |

**Table 4:** Results of bug finding between our approach, DPOR and SATCheck. ⊖ means the tool failed to run on the benchmark, ! the tool finished the exploration without finding the bug, $*$ the tool repeats the same execution and did not terminate. Since SATCheck does not support PSO, we only report its results on SC and TSO.

| **Program** | **DPOR** | | | **SATCheck** | | **MCR (our approach)** | | |
|---|---|---|---|---|---|---|---|---|
| | SC | TSO | PSO | SC | TSO | SC | TSO | PSO |
| **Dekker** | 22 | 28 | 29 | 32$^*$ | 68735$^*$ | 10 | 4 | 5 |
| **Lamport** | 6 | 8 | 24 | ⊖ | ⊖ | 2 | 2 | 3 |
| **Bakery** | 12 | 15 | 15 | ⊖ | ⊖ | 8 | 8 | 15 |
| **Peterson** | 4 | 5 | 6 | 19! | 34282$^*$ | 7 | 2 | 3 |
| **StackUnsafe** | 6 | 6 | 6 | ⊖ | ⊖ | 2 | 2 | 2 |
| **RVExample** | 301 | ⊖ | ⊖ | 60564$^*$ | 70365$^*$ | 53 | 54 | 39 |
| **Example** | 14840! | 14840! | ⊖ | 1! | 1! | 2767! | 2767! | 3 |

**Table 5:** Results of MCR under SC, TSO and PSO on real programs for state-space exploration and bug finding.

| Program | **#Executions** | | | **# Data Races** | | |
|---|---|---|---|---|---|---|
| | SC | TSO | PSO | SC | TSO | PSO |
| Account | 7 | 12 | 12 | 3 | 3 | 3 |
| Airline | 8 | 11 | 11 | 0 | 0 | 0 |
| StringBuf | 3 | 3 | 3 | 0 | 0 | 0 |
| Allocation | 30 | 30 | 30 | 0 | 0 | 0 |
| PingPong | 411 | 483 | 527 | 7 | 7 | 7 |
| Weblech | 178 | 103 | 116 | 6 | 6 | 6 |

lead to runtime exceptions are revealed faster under TSO and PSO.

## 7. RELATED WORK

Stateless model checking (SMC) prevails since the pioneering work of VeriSoft [17]. Since then a large research effort has been dedicated to reduction techniques that alleviate the state explosion problem. The most popular techniques known are Partial Order Reduction (POR) [12, 16] and context bounding [31, 32], while context bounding does not reduce redundancy but limits the search space to polynomial. A number of techniques [3, 13, 31] based on POR or combining them have been proposed to improve and optimize the performance of POR. However, as pointed out in MCR [21], the effectiveness of POR is limited by *happens-before*: it cannot reduce redundant interleavings that have different *happens-before* relations. Although MCR can maximally reduce the redundant exploration, it assumes SC and does not allow reordering of operations out of program order. The key conceptual contribution of this paper is enhancing MCR to support TSO and PSO by solving two challenge problems: relaxed *happens-before* modeling, and replay.

The feasibility of verifying concurrent programs under relaxed memory models have been studied before [6, 7, 9]. Abdulla et al. [4] apply SMC techniques to TSO and PSO by adopting a chronological trace presentation to relax the behavior of SC. Similar to [4], Zhang et al. [38] develop an approach that extends the original DPOR algorithm [16] to support TSO and PSO. The approach refines the dependent set to allow the reordering and introduces shadow threads to simulate the non-determinism of independent events by each thread. Both of the two approaches leverage DPOR to reduce the state space. However, since DPOR is limited by the *happens-before* relation, these approaches are less effective than MCR.

Several constraint-based approaches have also been proposed for verifying concurrent programs, including CheckFence [8], SATCheck [15] and MemSAT [36]. CheckFence verifies concurrent data structures for relaxed memory models by explicitly encoding all relevent events into a boolean formula. MemSAT verifies various weak memory models by specifying the memory model as a set of constraints in relational logic. By solving the constraints that encode both the memory specifications and the program assertions, it is able to find subtle bugs in test programs that satisfy the constraints.

There also exist several hybrid techniques [10, 11] that combine stateless model checking with testing. Sober [10] develops a run-time monitoring algorithm to detect violations of SC by exploring SC-only executions and checking their correctness via a model checker. Similar to our approach, RELAXER [11] replays the program under an active scheduler that enforces the semantics of relaxed memory models by delaying selected writes or reads. However, as a testing tool, RELAXER is unsound that it does not explore all state-space and it may miss bugs.

## 8. CONCLUSION

We have presented an extension of MCR for stateless model checking of concurrent programs under TSO and PSO. Our approach solves two key technical challenges. First, how to generate new unique interleavings by formulating the operational semantics of TSO and PSO as first-order logical constraints. Second, how to deterministically execute the program following the generated TSO and PSO interleavings. By relaxing the must happen-before constraints in MCR to allow TSO and PSO reorderings, and by developing novel replay algorithms that allow executions out of program order, our approach enables MCR to effectively verify concurrent programs for TSO and PSO. We have also presented our experimental results of applying MCR on both popular benchmarks and real applications and comparing MCR with DPOR and SATCheck. Our results show that our approach is much more effective than the other approaches for both state-space exploration and bug finding.

## Acknowledgement

The authors would like to thank Lawrence Rauchwerger and the anonymous reviewers for their constructive comments on earlier versions of this paper. This research was supported by faculty start-up funds from Texas A&M University and NSF award CCF-1552935.

## References

[1] ASM bytecode analysis framework. `http://asm.ow2.org/`.

[2] A real-world bug caused by relaxed consistency. `http://stackoverflow.com/questions/16159203/`.

[3] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[4] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas. Stateless model checking for TSO and PSO. *CoRR*, 2015.

[5] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[6] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[7] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models? In *Programming Languages and Systems*, pages 26–46. Springer, 2012.

[8] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[9] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[10] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer Aided Verification*, pages 107–120. Springer, 2008.

[11] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 122–132. ACM, 2011.

[12] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[13] K. E. Coons, M. Musuvathi, and K. S. Mckinley. Bounded partial-order reduction. In *In Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 833–848, 2013.

[14] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] B. Demsky and P. Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015.

[16] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[17] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997.

[18] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 2005.

[19] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[20] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[21] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. *In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[22] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[23] S. International. *The SPARC Architecture Manual: Version 8*. 1992.

[24] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[25] A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *Proceedings of the 17th International SPIN Conference on Model Checking Software*, SPIN'10, 2010.

[26] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *Proceedings of the 18th International SPIN Conference on Model Checking Software*, 2011.

[27] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 533–536, 2007.

[28] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2005.

[29] A. Mazurkiewicz. Trace theory. In *Petri nets: applications and relationships to other models of concurrency*, pages 278–324. Springer, 1986.

[30] T. Mitra, A. Roychoudhury, and Q. Shen. Impact of Java Memory Model on Out-of-Order Multiprocessors. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2004.

[31] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical report, MSR-TR-2007-12, Microsoft Research, 2007.

[32] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, pages 267–280, 2008.

[33] S. Owens, S. Sarkar, P. Sewell, and A. Better. x86 Memory Model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, 2009.

[34] A. Roychoudhury. Formal reasoning about hardware and software memory models. In *Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM, 2002.

[35] T. F. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for sequentially consistent systems. In *Runtime Verification*, pages 136–150. Springer, 2013.

[36] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking Axiomatic Specifications of Memory Models. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[37] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN, 2008.

[38] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. *In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.