Dynamic Partial Order Reduction for Relaxed Memory Models

Naling Zhang
Virginia Tech
Blacksburg, VA, USA
naling@vt.edu

Markus Kusano
Virginia Tech
Blacksburg, VA, USA
mukusano@vt.edu

Chao Wang
Virginia Tech
Blacksburg, VA, USA
chaowang@vt.edu

Abstract

Under a relaxed memory model such as TSO or PSO, a concurrent program running on a shared-memory multiprocessor may observe two types of nondeterminism: the nondeterminism in thread scheduling and the nondeterminism in store buffering. Although there is a large body of work on mitigating the scheduling nondeterminism during runtime verification, methods for soundly mitigating the store buffering nondeterminism are lacking. We propose a new dynamic partial order reduction (POR) algorithm for verifying concurrent programs under TSO and PSO. Our method relies on modeling both types of nondeterminism in a unified framework, which allows us to extend existing POR techniques to TSO and PSO without overhauling the verification algorithm. In addition to sound POR, we also propose a buffer-bounding method for more aggressively reducing the state space. We have implemented our new methods in a stateless model checking tool and demonstrated their effectiveness on a set of multithreaded C benchmarks.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Program; D.2.4 [*Software Engineering*]: Program Verification

Keywords Stateless model checking; partial order reduction; runtime verification; relaxed memory model; DPOR; TSO; PSO

1. Introduction

Shared-memory multiprocessors are increasingly common in today's computing systems. To achieve higher performance, they often implement memory models that are weaker than sequential consistency (SC) by employing optimizations such as speculative execution, buffering, and caching. Unlike the more intuitive SC model [22], where concurrent threads share a single memory that is always updated instantaneously by write operations, relaxed memory models can be more complex and sometimes non-intuitive [3], making program analysis and debugging difficult. For example, under the *Total Store Order (TSO)* model [30], exhibited by x86 processors, a write and a following read in the same thread, but from a different memory location, may be reordered. Under the *Partial Store Order (PSO)* model, which is a relaxation of TSO [40], two writes in the same thread, but to different memory locations, may also be reordered.

A promising technique for checking reachability properties of a concurrent program under SC is *stateless model checking* [18], which relies on systematic execution of the program to explore its state space and check whether properties hold at each state. Unlike their stateful counterparts which require the user to supply a finite-state model [14], stateless model checkers such as VeriSoft [18], CHESS [27], and Inspect [43] work directly on the software code written in languages such as C/C++, Java and C#, thereby making themselves more broadly applicable.

However, stateless model checkers suffer from the well-known *state explosion* problem, i.e., the size of the state space can be exponential in the size of the program. Although partial order reduction (POR) techniques [16] have been proposed for mitigating the scheduling nondeterminism, which is the main source of state explosion under SC, methods for mitigating the nondeterminism in store buffering under TSO and PSO are still lacking. In fact, many existing POR techniques would be subtly unsound when applied to relaxed memory models. That is, since they assume SC, a program verified by them as *bug free* could still exhibit buggy behaviors when run on microprocessors that implement TSO or PSO.

We propose a new dynamic partial order reduction method for soundly reducing the state space of a program running under TSO and PSO. Our method is sound for detecting deadlocks and assertion violations in a program with a finite and acyclic state space, which is consistent with existing POR methods [16]. It mitigates not only the scheduling nondeterminism but also the nondeterminism in store buffering. Under TSO and PSO, both types of nondeterminism can lead to an exponential growth of the state space. However, unlike thread scheduling, in previous stateless model checkers, it was not possible to model store buffering by varying the inter-thread order of concurrent operations. As a result, extending classic POR techniques from SC to TSO and PSO often requires an overhaul of the underlying verification procedure. In contrast, we model both types of nondeterminism in a unified framework, which allows classic POR techniques to be seamlessly applied to weaker memory models.

To illustrate the challenges in verifying programs under TSO, consider Figure 1, where the two threads share the variables x and y. The program can never print out $a = 0$ and $b = 0$ under SC. However, this is possible under TSO because the write of a_1 (or b_1) can be delayed past the following read of a_2 (or b_2). Existing tools such as VeriSoft [18], CHESS [27], and Inspect [43] cannot expose this buggy behavior because they assume SC. Furthermore, classic POR methods such as [16] would be unsound in reducing the state space of this program. More specifically, there are six possible interleavings under SC, among which three are redundant according to classic POR techniques. In contrast, under TSO there are twenty-four possible executions due to the additional nondeterminism in store buffering. We will show in Section 2 that twenty of them are

PLDI'15 , June 13–17, 2015, Portland, OR, USA

DOI: http://dx.doi.org/10.1145/(to come)

```
int x = y = 0;
thread1() {
  int a;
  x = 1;     //W(x) a1
  a = y;     //R(y) a2
  printf("a=%d\n", a);
}
thread2() {
  int b;
  y = 1;     //W(y) b1
  b = x;     //R(x) b2
  printf("b=%d\n", b);
}
```

Figure 1. A TSO example.

```
   int x = y = 0;
1  thread1() {
2    x = 1;      //W(x) a1
3    y = 1;      //W(y) a2
4  }
5
6
7  thread2() {
8    if(y== 1){//R(y) b1
9      if(x==0) //R(x) b2
10       ERROR
11   }
12 }
```

Figure 2. A PSO example.

redundant and therefore can be skipped by our new dynamic partial order reduction method.

To illustrate the challenges in verifying programs under PSO, consider Figure 2, where the two threads use y as a communication flag. Since the second thread checks the flag before checking whether x is set to 0 at Line 9, the ERROR label is not reachable under SC or TSO. However, under PSO, due to the use of separate store buffers for x and y, the two writes may take effect in a reverse order, making ERROR reachable. Under both SC and TSO, there are only three possible executions, but under PSO, there are seven. We will show in Section 4.4 that our new method can reduce the number of executions from seven down to three.

Our new method models nondeterminism in both thread scheduling and store buffering in a unified framework, by dynamically relaxing the *enabled* set in the DPOR algorithm [16] to capture the reordering of intra-thread transitions. This is the main difference from existing works such as Nidhugg [2] and CDSchecker [29]. In Nidhugg, Abdulla et al. proposed a stateless model checking algorithm for TSO and PSO, which relaxed a source-DPOR algorithm [1] by replacing the classic notion of *Mazurkiewicz traces* with a new and canonical partial order representation called *chronological traces*. In CDSchecker, Norris and Demsky implemented a systematic testing algorithm for concurrent data structures running under the C++11 relaxed memory model, which is different from TSO and PSO. Our method differs from these works in its unified modeling of the two sources of nondeterminism, which allows both persistent set and sleep set based POR techniques to be extended from SC to PSO and TSO.

In addition to sound POR optimization, we also propose a *buffer bounding* (BB) method to more aggressively reduce the state space while retaining the bug-finding capability as much as possible. Given a bound on the store buffer size, this method will explore only those TSO/PSO runs that are feasible under fixed-size store buffers. Whenever a store buffer is full, a new write would force the buffer to flush immediately, thereby reducing the nondeterminism in store buffering. This new method is analogous to, but also independent of *context bounding (CB)* [8, 27, 32], a popular method for mitigating the nondeterminism in thread scheduling.

We have implemented our new POR and BB methods in a tool built upon the stateless model checker Inspect [43], for which we also developed a front-end using the Clang/LLVM compiler to handle multithreaded C/C++ applications using PThreads. We have conducted an experimental evaluation on a large set of benchmark programs, including 121 litmus tests for x86-TSO and 15 multithreaded programs from SV-COMP [36]. The results show that our methods are effective in detecting TSO/PSO related failures and efficient in reducing the state space.

To sum up, we make the following contributions:

1. We propose a dynamic partial order reduction method for soundly reducing the state space during runtime verification of a concurrent program under TSO/PSO.
2. We also propose a heuristic method based on *buffer bounding*, which is a supplemental reduction technique aiming to quickly find violations rather than verifying their absence.
3. We implement these new methods in a runtime verification tool for stateless model checking of multithreaded programs.
4. We conduct experiments on a set of C programs to demonstrate the effectiveness of the proposed methods.

The remainder of this paper is organized as follows. First, we use examples in Section 2 to motivate our work. Then, we establish the notation in Section 3, before presenting our new method in Section 4. We present our *buffer-bounding* optimization in Section 5. We present our experimental results in Section 6, review related work in Section 7, and finally give our conclusions in Section 8.

2. Overview

In this section, we explain how we model the two types of nondeterminism in a unified framework, before illustrating how our method works on the running example in Figure 1.

2.1 Modeling Nondeterminism

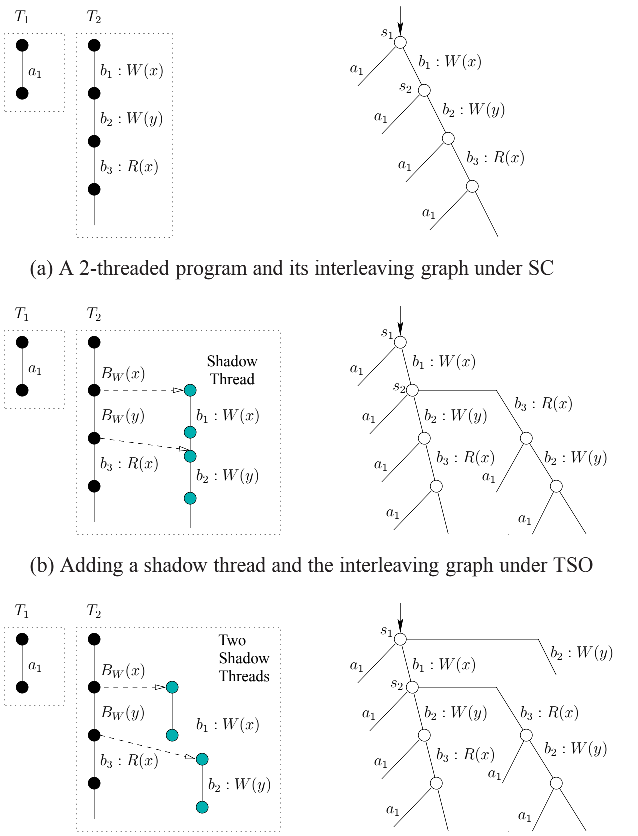
Our idea is to model the two types of nondeterminism using the same *interleaving graph*, which forms the foundation for analyzing POR methods and allows them to seamlessly be extended from SC to TSO and PSO.

Consider the two threads on the left-hand side of Figure 3 (a), where thread T_2 writes to x and y before reading from x. Under SC, there are four possible interleavings between a_1 in thread T_1 and $b_1 \ldots b_3$ in thread T_2, as shown in the graph on the right-hand side. Here, each node represents a global control state, i.e., a combination of each thread's program location, and each edge represents an instruction in either thread. For example, from the initial state s_1, one can execute a_1 from thread T_1, or b_1 from thread T_2. If b_1 is executed, we move to the state s_2, where either a_1 or b_2 can be executed. Under SC, we say that a_1 and b_1 are enabled at s_1, denoted $enabled_{SC}(s_1) = \{a_1, b_1\}$. Similarly, we have $enabled_{SC}(s_2) = \{a_1, b_2\}$.

To model the *per-thread* store buffer in TSO, we imagine that each thread has a *shadow thread* running concurrently and sharing a store buffer with the original thread. Each write in the original thread is implemented as two elementary operations: a *buffer-write* (B_W) by the original thread, followed by a *memory-write* (W_τ) by the shadow thread. As shown on the left-hand side of Figure 3 (b), there is a causal order (must-happen-before) between B_W and the corresponding W_τ. Furthermore, there is a causal order between $B_W(x)$ and $R(x)$ in the original thread to ensure the read-from-own-write requirement of TSO, i.e., a read (b_3) always gets the most recent value written by the same thread (b_1).

Now, the program's behavior under TSO can be characterized by the set of all possible interleavings of the original and shadow threads, subjecting to the causal ordering edges. This method for modeling store buffering directly follows the definition of TSO, where data in the store buffer are flushed to the main memory nondeterministically (as modeled by the shadow thread). Since B_W operations are local to the original thread (e.g., thread T_2's $B_W(x)$ and $B_W(y)$ are *invisible* to thread T_1), they are omitted in the new interleaving graph shown on the right-hand side. As a result, only a_1 and b_1 can be executed in state s_1, but both b_2 and b_3 can be executed in state s_2, together with a_1. That is, $enabled_{TSO}(s_1) = \{a_1, b_1\}$.

We say that the two types of nondeterminism are modeled uniformly because, from the new interleaving graphs alone, it is no longer possible to distinguish edges from the scheduling nondeterminism or store-buffer nondeterminism. As a result, during stateless model checking, where the goal is to systematically explore all paths of the interleaving graph, we do not need to distinguish one type of edge from another. This is the reason why classic POR



(a) A 2-threaded program and its interleaving graph under SC

(b) Adding a shadow thread and the interleaving graph under TSO

(c) Adding shadow threads and the interleaving graph under PSO

Figure 3. Modeling nondeterminism under SC, TSO, and PSO. Dashed edges are happens-before causality edges

methods, which are designed to prune away redundant paths in the interleaving graph under SC, can be extended using our method to TSO and PSO without significant modifications.

To model the per-address store buffers in PSO, we assume that there are multiple *shadow threads* correspond to each original thread, one for each memory address written by the original thread. This is shown on the left-hand side of Figure 3 (c). Since $B_W(x)$ and $B_W(y)$ write to different buffers, there is no longer a causal order between b_1 and b_2, thereby allowing their execution order to be reversed. With this in mind, we can construct the new interleaving graph shown on the right-hand side of Figure 3 (c). Compared to the interleaving graphs for SC and TSO, there are more paths allowed. For example, both of the two writes b_1 and b_2 can be executed at state s_1, denoted $enabled_{PSO}(s_1) = \{a_1, b_1, b_2\}$. Similarly, we have $enabled_{PSO}(s_2) = \{a_1, b_2, b_3\}$.

2.2 Partial Order Reduction for TSO/PSO

Knowing that the interleaving graphs for SC, TSO, and PSO capture all possible executions of the program under each memory model, we now explain how to soundly reduce the state space during stateless model checking.

Consider the running example in Figure 1. Under SC, there are six possible interleavings, three of which can be skipped by classic POR methods because they do not exhibit any additional behavior:

- $sc_1 = \circ \xrightarrow{a_1} \circ \xrightarrow{a_2} \circ \xrightarrow{b_1} \circ \xrightarrow{b_2} \circ$
- $sc_2 = \circ \xrightarrow{a_1} \circ \xrightarrow{b_1} \circ \xrightarrow{a_2} \circ \xrightarrow{b_2} \circ$
- $sc_3 = \circ \xrightarrow{a_1} \circ \xrightarrow{b_1} \circ \xrightarrow{b_2} \circ \xrightarrow{a_2} \circ$ (eqv. to sc_2; skip)

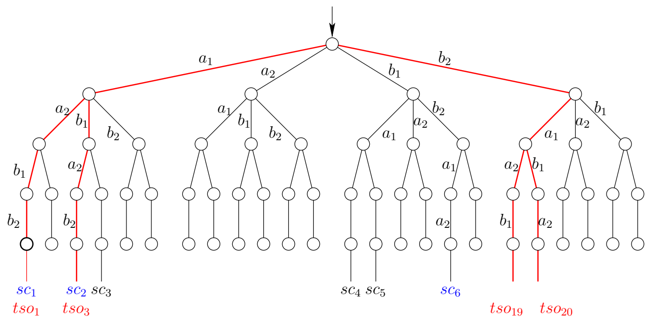


Figure 4. Example: valid interleavings of the program in Figure 1 under SC and TSO. There are 6 SC runs (three are redundant) and 24 TSO runs (twenty are redundant).

- $sc_4 = \circ \xrightarrow{b_1} \circ \xrightarrow{a_1} \circ \xrightarrow{a_2} \circ \xrightarrow{b_2} \circ$ (eqv. to sc_2; skip)
- $sc_5 = \circ \xrightarrow{b_1} \circ \xrightarrow{a_1} \circ \xrightarrow{b_2} \circ \xrightarrow{a_2} \circ$ (eqv. to sc_2; skip)
- $sc_6 = \circ \xrightarrow{b_1} \circ \xrightarrow{b_2} \circ \xrightarrow{a_1} \circ \xrightarrow{a_2} \circ$

Trace sc_3 is equivalent to sc_2 because changing the order of $b_2 : R(x)$ and $a_2 : R(y)$, which access different memory locations, does not affect the result. Similarly, traces sc_4 and sc_5 are equivalent to sc_2 because the order of a_1 and b_1 is immaterial. As a result, the stateless model checker only needs to explore sc_1, sc_2, and sc_6. In this case, classic POR methods [16] work well.

Under TSO, however, classic POR methods do not work nearly as well. As shown in Figure 4, the store buffer in thread T_1 can cause a_2 to be reordered before a_1, and the store buffer in thread T_2 can cause b_2 to be reordered before b_1. The combined impact of thread scheduling and store buffering would lead to twenty-four possible executions. Our new method will be able to systematically explore the state space and prune away the twenty redundant executions, thereby reducing the number of valid and irredudant TSO executions down to four, shown as follows:

- $tso_1 = \circ \xrightarrow{a_1} \circ \xrightarrow{a_2} \circ \xrightarrow{b_1} \circ \xrightarrow{b_2} \circ$ (same as sc_1)
- $tso_2 = \circ \xrightarrow{a_1} \circ \xrightarrow{b_1} \circ \xrightarrow{a_2} \circ \xrightarrow{b_2} \circ$ (same as sc_2)
- $tso_{19} = \circ \xrightarrow{b_2} \circ \xrightarrow{a_1} \circ \xrightarrow{a_2} \circ \xrightarrow{b_1} \circ$
- $tso_{20} = \circ \xrightarrow{b_2} \circ \xrightarrow{a_1} \circ \xrightarrow{b_1} \circ \xrightarrow{a_2} \circ$ (eqv. to sc_6)

Among these four, tso_{19} is not equivalent to any SC execution—it represents a new behavior unique to TSO.

It should not come as a surprise that tso_{20} is actually equivalent to sc_6, even though tso_{20} can never be produced under SC. They are equivalent under TSO because b_1 is independent of both b_2 and a_1, and by repeatedly swapping its order with respect to b_2 and a_1, we can transform sc_6 into the equivalent run tso_{20}; by Mazurkiewicz's trace theory [26] applied to our relaxed interleaving graph, they are equivalent.

During stateless model checking, we perform this equivalence based pruning on the fly. That is, we will explore only the four marked TSO executions shown in Figure 4 while having a guarantee of covering all possible TSO behaviors.

3. Preliminaries

This section provides the background information on memory models and stateless model checking.

3.1 Concurrent Systems

First, we introduce our model of a concurrent system. The system includes a finite number of threads communicating through a set of *shared objects*. A shared object can be any shared memory location, mutex lock, or condition variable. Each thread itself is a sequential program consisting of a finite number of statements. A statement on a shared object is said to be *visible*. Any other statement is considered to be *invisible*. Only one shared object can be accessed by a visible statement at time. Each visible statement is considered to be atomic. A statement can *block* if it cannot be executed due to the state of the program, e.g., when a mutex lock is held by a thread then all subsequent lock statements on the same lock will block.

We consider each dynamic execution of a statement in the program to be distinct. Let $stmts$ be the set of all statements in a program. Each execution of $st \in stmts$ is a *transition*. We represent a transition by the tuple $\langle tid, type, var, val \rangle$, where tid is the thread's ID, $type$ is the statement's type, var is the shared object being accessed, and val (optional) is the value used in the statement. A transition may have one of the following forms:

1. $\langle tid, load, var \rangle$ is a read from global variable var,
2. $\langle tid, store, var, val \rangle$ is a write of val to variable var,
3. $\langle tid, fork, var \rangle$ creates the child thread var,
4. $\langle tid, join, var \rangle$ joins the child thread var,
5. $\langle tid, lock, var \rangle$ acquires the lock var,
6. $\langle tid, unlock, var \rangle$ releases the lock variable var,
7. $\langle tid, wait, var \rangle$ waits on condition variable var, and
8. $\langle tid, notify, var \rangle$ wakes up a transition waiting on var.

A thread is *disabled* if it cannot executes the next statement. For example, a thread trying to acquire a mutex lock held by another thread, or waiting for a condition variable not yet set by another thread, or joining with a child thread not yet terminated, is disabled. If a thread is not disabled, it is *enabled*. Two transitions are *co-enabled* in a state s if they are both enabled in s.

We define a *state* in the system as the union of the states of all the threads. Since the concrete state (content of the shared memory) is not stored during stateless model checking, each state is uniquely identified by the sequence of transitions executed by all threads. The concurrent system is, formally, a transition system $A = (S, \Delta, s_0)$, where S is the set of all possible states, $\Delta \subseteq S \times S$ is the transition relation, and s_0 is the initial state. We use the notation $s \xrightarrow{t} s'$ to denote that executing the transition t from s leads to the state s'. We consider a state s' to be reachable from s if there exists a sequence of transitions starting from s and ending at s' ($s \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s'$). We assume that there are no cycles in the state space: executing a transition always creates a new state, which is consistent with the existing stateless model checkers.

3.2 Memory Models

The most intuitive memory model in concurrent systems is sequential consistency (SC) [22], which says that "the result of any SC execution is the same as if the operations of all the processors were executed in some sequential order and the operations of each individual processor appears in the sequence in the order specified by its program." That is, instructions from the same threads follow their order in the program (called the *program order*) and a write operation updates the memory instantaneously with respect to other memory reads (called *write-atomicity*). In contrast, TSO and PSO are deviations from SC by relaxing the *program-order* and *write-atomicity* requirements differently.

TSO can be viewed as a visible consequence of store buffering, where each processor has a FIFO buffer of the pending memory writes [33] to avoid blocking while a write completes. As a result, TSO relaxes the intra-thread program order by allowing a write and a following read from a different memory location to be reordered. Regarding the write-atomicity requirement, TSO allows a processor to read the value of its own most recent write from the store buffer, but prohibits it from reading the value of another processor's write before the write is made visible to all other processors, i.e., until the thread's store buffer is flushed to the memory.

PSO can also be viewed as a visible consequence of store buffering, where each processor has multiple FIFO buffers of the pending memory writes, potentially one buffer per memory address [40]. Regarding the intra-thread program order, PSO not only allows the reordering of a write and a following read, but also two consecutive writes to different memory addresses. Regarding the write-atomicity requirement, PSO allows a processor to read the value of its own most recent write from the FIFO buffer, but prohibits it from reading the value of another processor's write before the write is made visible to all other processors.

3.3 Stateless Model Checking

Stateless model checking [18] is a method to systematically explore the state space of a concurrent system. In contrast to its stateful counterpart [14, 19], a stateless model checker does not store the concrete states of the system. Instead, abstract states are used where each state is uniquely identified by the sequence of transitions executed starting from the initial state s_0—this is possible as long as each thread is a deterministic sequential program and the only source of nondeterminism comes from the thread interleaving. Therefore, instead of exploring the reachable states, the procedure systematically explores the set of execution traces of the system.

Partial order reduction (POR) techniques have been widely used in model checking, which group execution traces into equivalence classes and then explore at least one representative from each equivalence class. The soundness of POR relies on the notion of Mazurkiewicz trace [26], which formally defines the condition under which two traces are equivalent. Specifically, two transition sequences, ρ_1 and ρ_2, are equivalent if and only if ρ_1 can be obtained from ρ_2 by repeatedly permuting *independent* adjacent transitions. This is because, when two transitions t_1 and t_2 are independent, executing $t_1 t_2$ and $t_2 t_1$ leads to the same state.

Since the correctness of POR rests on the underlying dependency relation that defines the Mazurkiewicz trace, when extending the method from SC to TSO and PSO, we can switch one dependency relation with another without significantly modifying the stateless model checking algorithm. Under SC, two transitions are *dependent* if they are from the same thread, or if they are from different threads, access the same memory address, and at least one of them is a write. In contrast, under TSO and PSO, two transitions may be considered as independent even if they are from the same thread, e.g., a write and the following read from a different memory location. We shall expand the definition of the dependency relation from SC to TSO/PSO in the later sections.

Similar to many existing POR methods, the correctness of our method for pruning redundant interleavings will rest on the notion of *persistent sets*. A persistent set at a state s is a subset T of the transitions enabled at state s such that each transition not in T is independent with T. It has been proved [17] that exploring only transitions in the persistent set of each state guarantees detection of all reachability errors, such as deadlock and assertion violations, in a program with an acyclic state space. Below is a formal definition of the persistent set (cf. [17]).

DEFINITION 1. *A set T of transitions enabled in a state s is* persistent *iff for all nonempty sequences of transitions of the form*

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 ... \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1} \, ,$$

where $t_i \notin T$, $1 \leq i \leq n$, we have t_n is independent with all transitions in T.

Early POR methods statically computed the persistent set [17], which often leads to overapproximations due to limitations of the static analysis procedures, such as imprecise alias analysis. Dynamic POR (DPOR) [16] addressed this problem by dynamically computing the necessary transitions to explore using *backtrack sets*. Our work builds upon DPOR by extending it from SC to TSO and

PSO. As we will show in the next section, our extension does not fundamentally alter the algorithm, thereby allowing both persistent- and sleep-set based optimizations to be carried over.

4. Partial Order Reduction for TSO/PSO

We first generalize the baseline DPOR algorithm [16] for SC, and then present our extensions from SC to TSO and PSO.

4.1 Generalizing the DPOR Algorithm

Algorithm 1 shows a modification of the DPOR algorithm [16], where we model the two types of nondeterminism in a unified framework. The overall flow remains the same, but the definitions of *enabled*, *done*, and *backtrack* sets are expanded to account for the additional nondeterminism; when used for SC, the behavior of this modified algorithm remains the same as the original DPOR implementation.

Specifically, the *enabled* set at state s, in the original algorithm, was defined as the set of threads that can be executed at s. Under SC, since each thread is a deterministic program, it can execute at most one transition at any moment. Therefore, the *enabled* set of threads is equivalent to the set of immediate next transitions in these threads. The *done* set is a subset of the *enabled* transitions (threads) that have already been explored at state s—whenever *done* becomes the same as *enabled*, the state subspace starting from s has been fully explored. The *backtrack* set is a subset of the *enabled* transitions (threads) that is persistent with respect to the theory of partial order reduction.

Under TSO/PSO, however, a thread may execute more than one transition at any moment, e.g., the transition may be either a store-buffer flush or the following read in the same thread. To capture this nondeterminism, we modify the definitions of *enabled*, *done*, and *backtrack* sets. Instead of defining them as sets of threads (or transitions), we now define them as sets of *pairs of a thread and its transitions*, denoted $\{(tid, \langle transitions \rangle)\}$. For example, at the initial state in Figure 4, the enabled set under SC was $\{1, 2\}$, indicating that threads T_1 and T_2 can be executed. With our extension, the enabled set under SC and TSO are defined as follows:

- SC: $enabled_{SC}(s_1) = \{(1, \langle a_1 \rangle), (2, \langle b_1 \rangle)\}$
- TSO: $enabled_{TSO}(s_1) = \{(1, \langle a_1, a_2 \rangle), (2, \langle b_1, b_2 \rangle)\}$

While this is the only modification required to handle TSO and PSO, we first assume SC while reviewing the basics of DPOR. We delay the discussion of dynamically computing the enabled set, implemented in the subroutine UPDATEENABLEDSET (Line 3), until Section 4.4. Under SC, this subroutine has no effect.

Algorithm 1, starting from the initial state s_0, performs a depth-first search of the state space (A). During the depth-first search, the stack (S) contains a finite sequence of transitions: $t_0, t_1, \ldots, t_n$, which were each performed from the states $s_0, s_1, \ldots, s_{n-1}$. Therefore, S implicitly represents the execution trace $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_n} s_{n-1}$. As we have already mentioned, each state $s \in S$ has an *enabled* set, consisting of all transitions that may be executed from the state s. Each state s also has a *done* set, consisting of all enabled transitions that have been explored from the state. Each state s also has a *backtrack* set, which is the set of enabled transitions that still need to be explored from s. For ease of comprehension, the pseudocode uses the following notation:

- $dom(S)$ is the set $\{ 1, \ldots, n \}$ of indices,
- $pre(S, i)$ for $i \in dom(S)$ refers to the state s_i,
- $last(S)$ refers to s_{n+1},
- S_i is transition t_i (the i^{th} transition in S),
- $next(s, p)$ is the set of transitions (unique under SC but not TSO/PSO) to be executed by thread p in state s,
- $S.t$ appends transition t to S, and
- $thd(t)$ is the thread that executed the transition t.

As in the original DPOR, we define a happens-before relation on transitions in $S = t_1 \ldots t_n$, denoted $i \rightarrow_S t$ for any transition t and index $i \in dom(S)$. It is the smallest relation on the set $\{ 1 \ldots n \}$ such that:

1. if $i \leq j$ and t_i is dependent with t_j, then $i {\rightarrow_S} t_j$
2. $\rightarrow_S$ is transitively closed.

Recall that under SC, the transition t_i is always dependent with t_j if they are from the same thread, or if they are from different threads, access the same memory address, and at least one of them is a write. We will relax the dependency relation for PSO and TSO in the next subsection.

The DPOR algorithm for SC begins by calling $Explore()$ from the initial state. The stack (S) contains the sequence of transitions that have been executed to reach the current state $last(S)$. When arriving at a new state, the algorithm calls UPDATEBACKTRACK-INFO(), which checks the next transition of each thread (t_n) to search the stack for the last transition (S_i) where

1. S_i is dependent and may be co-enabled with t_n, and
2. $i \not\rightarrow_S t_n$.

If a transition satisfying the requirements is found, the algorithm inserts a backtrack point in the state $pre(S, i)$. Intuitively, it aims to flip the order of these two transitions in a future run. Specifically, on Line 14, it attempts to find the subset E of enabled transitions in the state $pre(S, i)$ where each $t \in E$ happens-before t_n in S. The set E contains all such transitions in order to allow t_n to be executed before t in future runs. When no such transition is found (E is empty), the algorithm over-approximates by adding all the enabled transitions to the backtrack set.

It has been proved [16] that Algorithm 1 visits a set of transitions from each state which contains all the transitions in the persistent set of the state. Since the fundamental theorems proving the soundness of any POR method rest on the fact that the *persistent set* from each state is explored, Algorithm 1 is similarly able to soundly verify reachability properties (such as deadlocks and assertion violations) in a concurrent program with an acyclic state space.

4.2 Modeling Store Buffers in TSO and PSO

Next, we extend the *enabled*, *backtrack* and *done* sets in Algorithm 1 to handle relaxations in program order constraints caused by store-buffers from TSO and PSO.

To model TSO, we use the X86-TSO model created by Sewell et al. [33]. Specifically, we assume that a thread may interact with memory using the following operations:

- $R(x)$, a read of variable x,
- $B_W(x)$, a write to variable x in the thread's store-buffer,
- $W_\tau()$, a dequeue of the oldest write in the store-buffer.

$R(x)$ is defined in the same manner as before (Section 3.1). $W_\tau()$ is the nondeterministic flushing of a store-buffer, which may occur at anytime when the thread is not disabled. In the context of DPOR, $B_W(x)$ is an invisible operation since it only affects the thread's store-buffer and does not need to be monitored. In the end, this model adds one transition type to our state system: $W_\tau()$.

To model PSO, we use the same operations as TSO except for changing $W_\tau()$ to $W_\tau(x)$ for some variable x. This operation dequeues the oldest write to x by the thread. Intuitively, both definitions—$W_\tau()$ and $W_\tau(x)$—match the semantics of having one store buffer in TSO and multiple store buffers in PSO. In both cases, we consider the store-buffer(s) to be of infinite length, i.e., a write to memory could be delayed indefinitely.

For both TSO and PSO, our stateless model checking system transforms each shared memory write in the program into a store-buffer write B_W. At the next state, the thread will have an enabled $W_\tau()$ or $W_\tau(x)$ transition in addition to its next program transition.

Algorithm 1 Modified DPOR [16] to allow for TSO/PSO behavior. Under SC, the behavior is equivalent to the original implementation.

Initially: Explore($\{s_0\}$)
function EXPLORE(S)
  $s \leftarrow last(S)$
  UPDATEENABLEDSET(S, s) ▷ This call has no effect under SC. See Algorithm 2.
  UPDATEBACKTRACKINFO(S, s)
  if $\exists t \in enabled(s)$ **then**
    $backtrack(s) \leftarrow \{t\}$
    $done \leftarrow \varnothing$
    while $\exists t \in (backtrack(s) \setminus done)$ **do**
      add t to $done$
      Explore($S.t$)
function UPDATEBACKTRACKINFO(S, s_n)
  let $dom(S)$ be the set $\{\ 1, \ldots, n\ \}$ of indices of states in S up to s_n
  for all transitions $t_n \in next(s, p)$ of all threads p **do**
    if $\exists i = max(\{i \in dom(S) \mid S_i$ is dependent and may be co-enabled with t_n and $i \not\to_S t_n\})$ **then**
      $E \leftarrow \{t \in enabled(pre(S, i)) \mid thd(t) = thd(t_n)$ or $\exists j \in dom(S) : j > i$ and $t = S_j$ and $j \to_s t_n\}$
      if $E \neq \varnothing$ **then**
        add any $t \in E$ to $backtrack(pre(S, i))$
      else
        add all $t \in enabled(pre(S, i))$ to $backtrack(pre(S, i))$

A thread will always have a $W_\tau()$ or $W_\tau(x)$ transition enabled until its store-buffer(s) are empty.

For example, consider a thread executing the following statements: `x = 5; if (z > 5)`. After executing `x = 5`, the store-buffer, under TSO, will have `5` enqueued into it. At the next state, the thread will have two enabled transitions: $W_\tau()$ and $R(z)$. Intuitively, the $W_\tau()$ operation can be thought of as a shadow thread, as described in Section 2, which can dequeue the store-buffer at anytime. But if the read of x is the next program statement in the thread, TSO also guarantees that $R(x)$ gets the value 5 written by $W_\tau()$.

Under PSO, the procedure behaves similarly except that, instead of a single $W_\tau()$ operation for each thread, there can be one $W_\tau(x)$ operation for each variable x in a thread.

Fence instructions can be modeled by repeatedly executing $W_\tau()$ or $W_\tau(x)$ operations until the buffer(s) are empty. The program can execute a fence instruction explicitly, e.g., when such instruction exists in the program code, or implicitly, e.g., when it executes synchronization primitives such as $lock()$ and $unlock()$ that also force the entire store-buffer(s) of the thread to flush.

4.3 Relaxing the Intra-thread Dependency Relation

Armed with the new $W_\tau()$ and $W_\tau(x)$ operations, we now extend the *enabled*, *done*, and *backtrack* sets from SC to PSO and TSO to capture the potentially multiple enabled transitions from each thread. Here, the $W_\tau()$ operation is considered in the same way as a memory write, since we know the value to be written to a given memory address when $W_\tau()$ is executed. Conveniently, with this extension, the semantics of Algorithm 1 do not change. The only difference is in the definition of the *intra-thread* dependency relation, which is not to be confused with the *inter-thread* dependency relation. Recall that the latter is defined as follows: two transitions from different threads are dependent iff they access the same memory and at least one of them is a write. We do not change the definition of the inter-thread dependency relation.

DEFINITION 2. *Under SC, two transitions t_1 and t_2 executed by the same thread are always (intra-thread) dependent, denoted $(t_1, t_2) \in \mathcal{D}_{SC}$.*

For TSO and PSO, we choose to define the independence relation (as opposed to dependence), while assuming that all transitions that are not independent are dependent.

DEFINITION 3. *Under TSO, two transitions t_1 and t_2 executed by the same thread are independent, denoted $(t_1, t_2) \notin \mathcal{D}_{TSO}$, iff t_1 is a write, t_2 is a following read, $addr(t_1) \neq addr(t_2)$, and for all t_3 in between t_1 and t_2 (if any), t_3 must be a write such that $addr(t_3) \neq addr(t_2)$.*

DEFINITION 4. *Under PSO, two transitions t_1 and t_2 executed by the same thread are independent, denoted $(t_1, t_2) \notin \mathcal{D}_{PSO}$, iff t_1 is a write, t_2 is a following read or write, $addr(t_1) \neq addr(t_2)$, and for all t_3 in between t_1 and t_2 (if any), t_3 must be a write such that $addr(t_3) \neq addr(t_2)$.*

For a given dependency relation $\mathcal{D}$ (which can be $\mathcal{D}_{SC}$, $\mathcal{D}_{TSO}$, and $\mathcal{D}_{PSO}$), the enabled set consists of, for each thread, its immediate next transition t_1 as well as all following transitions t_2 such that $(t_1, t_2) \notin \mathcal{D}$. Therefore, relaxing the dependency relation from SC to TSO and PSO lead to the expansion of the *enabled* set (and hence the *done* and *backtrack* sets), meaning that previously sequential transitions within a thread may be reordered. This is the reason why for the initial state s_1 of the program in Figure 2, whose interleaving graph is in Figure 5, the enabled set under SC and TSO are $\{(1, \langle a_1 \rangle), (2, \langle b_1 \rangle)\}$, whereas $enabled_{PSO}(s_1) = \{(1, \langle a_1, a_2 \rangle), (2, \langle b_1 \rangle)\}$.

Given that our definitions of $\mathcal{D}_{TSO}$ and $\mathcal{D}_{PSO}$ directly follow the X86-TSO model [33] and the SPARC manual [40], the expanded enabled set precisely defines all possible TSO or PSO relaxations of program order.

4.4 Dynamically Updating the Enabled Set

There are two challenges in modifying the program order of a thread during DPOR as required by TSO and PSO. First, DPOR relies on dynamic execution of the program to discover, at each state s, the set of transitions that may be executed by each thread. It does not have access to the program paths that are not currently executed, which is required for computing a sound approximation of the enabled set. Second, since DPOR concretely executes each thread, which is a sequential program, it is difficult to perform the desired modifications to the program order at runtime due to limited visibility of future transitions.

To understand the issue of limited visibility, consider that a thread is about to execute `x = 5` which, in the SC model, is a write $W(x)$. At the current state s, the thread has an enabled transition which is a write to x. Under TSO, this write can potentially be re-ordered with any following read by this thread from a different memory address. As a result, we would like to enable any qualifying read along side the write to x. Under PSO, if after writing to x the thread writes to shared variable y then at the state s both the $write$ to x and the $write$ to y should be enabled allowing for their order to be permuted. However, during DPOR, each thread is running in the native OS environment, executing one transition at a time, and only the next statement to be executed by each thread can become visible to the model checker (e.g., the $write$ to y cannot be seen before the write to x is executed).

Our method for dynamically updating the *enabled* set is similar, in spirit, to the method for dynamically updating the *backtrack* set used in DPOR [16], where the authors faced a similar problem in computing the backtrack set under SC: statically computing the set means that it often has to be drastically overapproximated, which leads to missed opportunities for optimization, whereas dynamically updating the set retroactively at run time leads to precise results. Inspired by the idea, we propose to retroactively update the *enabled* set for TSO and PSO during run time.

Our procedure for retroactively updating the *enabled* set is invoked on Line 3 in Algorithm 1. At this moment, we initialize the enabled set for state s to include the immediate next transitions from all non-blocking threads. Then, we traverse the stack S backwardly to update the enabled sets of the preceding states. If any transition enabled in s can potentially be reordered with transitions in a preceding state s' due to program order relaxations, we will update the enabled set of s' recursively. In addition, we will recompute the backtrack information using the new enabled set of state s' by invoking the subroutine UPDATEBACKTRACKINFO.

Algorithm 2 shows the pseudocode to dynamically update the enabled set of each state. Similar to updating the backtrack set during DPOR, this procedure scans the state stack (S) to find transitions which are *intra-thread* independent based on one of the previous definitions ($\mathcal{D}_{SC}$, $\mathcal{D}_{TSO}$, and $\mathcal{D}_{PSO}$). Since under SC the program order cannot be modified this procedure does nothing, whereas under TSO and PSO, it expands the enabled set to include any future transition that should be included. We use $enabled(s_i, p)$ to represent the enabled set of the i^{th} state in S for the p^{th} thread.

Algorithm 2 Procedure to dynamically update the enabled set of each thread when testing under TSO/PSO.

```
function UPDATEENABLEDSET(S, s_n)
    if Testing under SC then
        return
    let D be either D_PSO or D_TSO depending on the Testing mode
    modified ← { }
    let n be the index of last(S)
    for all transitions t_n = next(s_n, p) of all threads p do
        for all j = n − 1, . . . , 1 do
            for all transitions t_j = next(s_j, p) do
                if (t_n, t_j) ∉ D then
                    enabled(s_j, p) ← enabled(s_j, p) ∪ { t_n }
                    modified ← modified ∪ { s_j }
    for all states s_j ∈ modified do
        UPDATEBACKTRACKINFO(S, s_j)
```

The algorithm takes the current stack S as input and examines the predecessors of last state in S, denoted $s_n = last(S)$. For any predecessor state s_j, the algorithm checks if there are two transitions t_j and t_n from the same thread p such that they may be permuted (i.e., they are *intra-thread* independent). If such transitions are found, then the enabled set of s_j for thread p, denoted $enabled(s_j, p)$, is updated to include the transition t_n. Additionally, whenever the enabled set of a predecessor state is updated, its backtrack set is also updated, by invoking UPDATEBACKTRACK-INFO. This ensures that the newly enabled transitions can be potentially permuted in future runs. The entire process terminates after traversing the stack S once backwardly.

Consider the example in Figure 2, which has seven PSO-compatible runs (three are also SC runs) as shown in Figure 5. Our method first explores $pso_1 = \circ \xrightarrow{a_1} \circ \xrightarrow{a_2} \circ \xrightarrow{b_1} \circ \xrightarrow{b_2} \circ$.

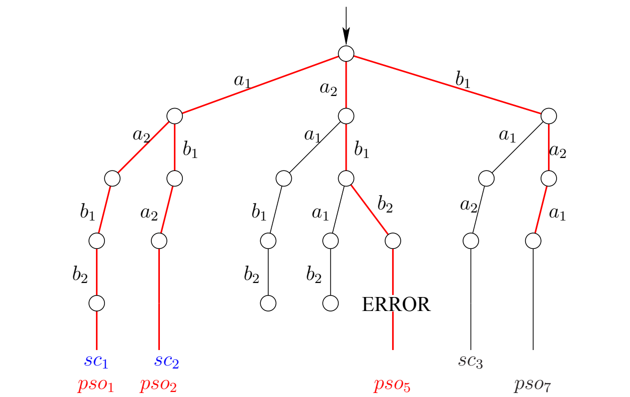


Figure 5. The seven PSO-compatible runs for the program in Figure 2, where four runs are redundant and skipped.

When it reaches state s_1 for the first time, $enabled_{PSO}(s_1)$ is set to $\{(1, \langle a_1 \rangle), (2, \langle b_1 \rangle)\}$ meaning that a_1 and b_1 are the immediate next transitions to be executed in threads T_1, and T_2 respectively. After reaching s_2, however, we discover that a_2 follows a_1 in thread T_1. Since $(a_1, a_2) \notin \mathcal{D}_{PSO}$, we go back to state s_1 and retroactively update its enabled set to $enabled_{PSO}(s_1) = \{(1, \langle a_1, a_2 \rangle), (2, \langle b_1 \rangle)\}$. As a result, eventually pso_5 will be explored, which represents a behavior that is unique to PSO, i.e., not possible under SC or TSO. Specifically, in pso_5, the second write $(y = 1)$ in Figure 2 is flushed to the memory before the first write $(x = 1)$, making the ERROR label at Line 10 reachable. Also note that our POR method executes only pso_1, pso_2, and pso_5 while completely skipping the other four PSO runs because they are equivalent to the three explored runs.

THEOREM 1. *Our $DPOR_{TSO}$ algorithm explores all possible TSO-compatible execution traces and does not explore any TSO-incompatible execution trace.*

THEOREM 2. *Our $DPOR_{PSO}$ algorithm explores all possible PSO-compatible execution traces and does not explore any PSO-incompatible execution trace.*

Since we lift the original DPOR algorithm from SC to TSO and PSO by relaxing the intra-thread dependency relation D_{SC}, which in turn leads to enlarged *enabled*, *done*, and *backtrack* sets in Algorithm 1, here, we only need to show that the relaxations are correct. Since our definitions of D_{TSO} and D_{PSO} directly follow the X86-TSO model [33] and the SPARC manual [40], and the original DPOR algorithm is known to be sound in pruning redundant executions [16], the above two theorems hold.

5. Buffer Bounding Based Analysis

Under SC, an important idea for reducing the complexity of concurrency testing is *context bounding (CB)* [8, 27, 32], which explores only executions with a bounded number of preemptive context switches. Although this is an unsound reduction in that it may miss valid program behaviors, empirical studies show that it is effective in detecting real bugs, which tend to involve few context switches. However, context bounding only focuses on mitigating state explosion caused by the scheduling nondeterminism. We propose a new *buffer bounding* method to mitigate the state explosion caused by the nondeterminism in store buffering. As such, it is complementary to the existing methods on context bounding.

Our default model for TSO/PSO assumes that a write to memory may be delayed indefinitely. We propose bounding the size of the store buffers to cut down the search space. The goal is to speed up testing (quickly finding bugs) as opposed to verification (proving

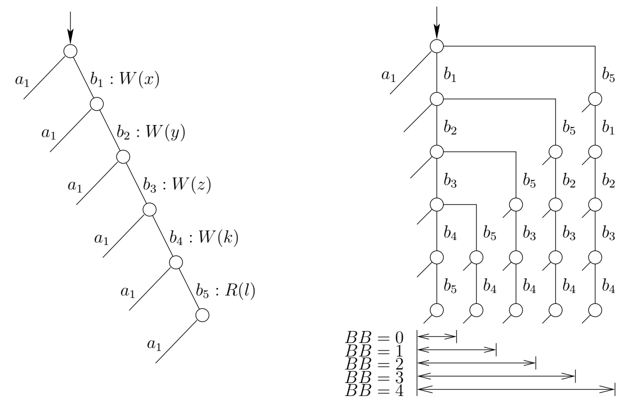


Figure 6. Reduction in the number of TSO runs with the buffer bound (BB) set to 0, 1, 2, 3, and 4, respectively.

the absence of bugs). Whenever a store-buffer of a thread is full, executing a write operation will force it to flush immediately, to make room for the new write.

Under the extreme case where the buffer size is set to 0, our TSO/PSO models would degenerate into SC. Under the other extreme case where the buffer size is set to $+\infty$, our TSO/PSO models would conform to the TSO/PSO standard. In between 0 and $+\infty$, the set of runs explored by our method would be a superset of the runs explored under SC and a subset of the runs explored under TSO/PSO.

Figure 6 shows an example where one thread executes a_1 and the other thread executes the sequence of instructions $b_1 b_2 b_3 b_4 b_5$. The interleaving graph under SC is shown on the left-hand side, whereas the interleaving graph under TSO is shown on the right-hand side. Under TSO, the read in b_5 may be reordered with all preceding writes. However, if we bound the store-buffer size to one, only two of the five permutations, illustrated on the right-hand side of Figure 6, will be allowed, since b_5 cannot only be reordered before b_3. If we bound the store-buffer size to 2, b_5 can be reordered before both b_4 and b_3, which leads to more valid interleavings.

Although the idea of buffer bounding has been exploited before in other contexts [23, 24], its effectiveness have not been experimentally evaluated in stateless model checking. We fill the gap by providing the first implementation as well as experimental evaluation in the following section.

6. Experiments

We have implemented our new methods, along with the original DPOR [16] with sleep-set reduction for SC, in a software verification tool called *rInspect*. The tool builds upon Inspect [43] and the popular Clang/LLVM compiler for handling C/C++ code written for the Linux/PThreads platform. We have conducted experiments on a large set of publicly available benchmarks. Our evaluation was designed to answer two research questions:

- Is our unified framework for handling both scheduling and buffering nondeterminism effective in detecting TSO/PSO related violations? Is it effective in reducing the search space?
- We proposed buffer-bounding to more aggressively reduce the search space while retaining the bug-detection capability as much as possible. Is it effective in practice?

Our benchmarks include 121 small programs, consisting of both the litmus tests for x86-TSO [4] and nine concurrent C programs from various prior publications [9, 10], which implement low-level concurrency protocols. Additionally, we used various versions of 15 multithreaded programs from the concurrency section of the

Table 1. Results on the x86-TSO litmus test programs.

Method	Passed	Failed	Avg. # Runs
DPOR_{SC}	121	0	1.0 X
DPOR_{TSO}	47	73	5.0 X
DPOR_{PSO}	24	97	11.0 X

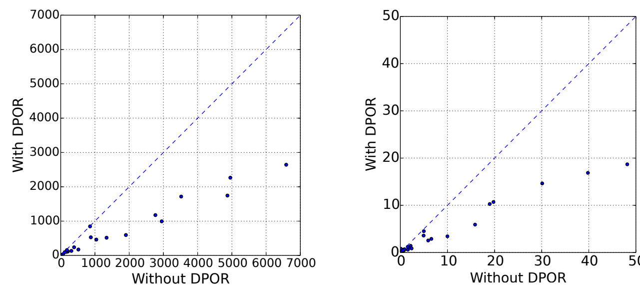


Figure 7. Reduction in the number of runs (left) and execution time in seconds (right) for DPOR_{TSO} on SV-COMP programs.

Software Verification Competition (SV-COMP) [36]. All experiments were obtained from a desktop with Intel Core i5-3340 3.10 GHz CPU running 32-bit Linux.

First, we evaluate the effectiveness of our method for detecting assertion violations. Table 1 shows the results of running the method on the x86-TSO litmus test programs. For all benchmark programs, our tool has correctly verified the program or detected the violation. For each memory model, we report the number of passing (violation free) and failing programs, as well as the normalized average number of runs explored by each method. The normalized number of runs for DPOR_{SC} is 1. For DPOR_{TSO}, there are on average five times more runs to be explored. For DPOR_{PSO}, there are eleven times more runs.

The results show that the baseline algorithm DPOR_{SC}, while having a lower number of runs, significantly under-approximates the behavior of the program under TSO and PSO. As a result, it would miss many violations. That is, DPOR_{SC} may claim the program as violation free when in fact under TSO and/or PSO the program has a violation.

Next, we show the effectiveness of our method in pruning redundant runs. Toward this end, we compare the number of runs explored, and the time taken, by the stateless model checker with and without our new POR method. Figure 7 shows the results of running the SV-COMP benchmark programs in two scatter plots, where the x-axis represents the number of runs (and the time in seconds) of the baseline stateless model checker under TSO, and the y-axis represents the same data for DPOR_{TSO}. Figure 8 shows the same type of scatter plots, but for DPOR_{PSO}. In these figures, each point below the diagonal line is a winning case for our method. These results show that our new POR method significantly reduces the number of runs and the execution time.

Finally, we evaluate the effectiveness of buffer bounding in reducing the search space while retaining the failure-detection capability as much as possible. Table 2 shows the number of violations detected from all benchmark programs by using a bounded TSO/buffer of size 0, 1, 2, 3, ..., $+\infty$. The results show that, for the set of benchmark programs we used, most of the TSO related violations can be detected using the buffer bound 2 (92% detection rate) or 3 (97% detection rate).

Figure 9 shows the number of runs explored by TSO/buffer bounding on a parameterized *Dijkstra* program from SV-COMP. As we gradually increase the number of concurrent operations in the program, we recorded the growth rate in the number of runs explored by DPOR_{TSO} under different buffer bounds. The results

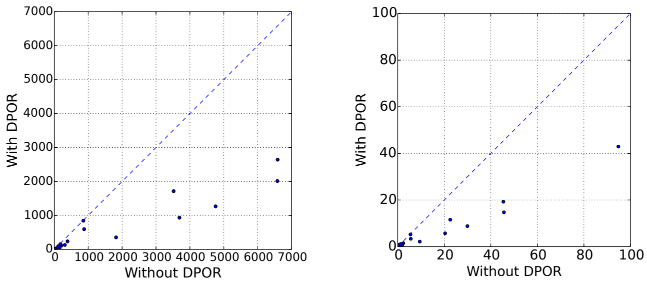


Figure 8. Reduction in the number of runs (left) and execution time in seconds (right) for DPOR$_{PSO}$ on SV-COMP programs.

Table 2. Bugs detected by DPOR$_{TSO}$ with buffer bounding.

	BB=0 (SC)	BB=1	BB=2	BB=3	BB=$+\infty$
Bugs (%)	0.0	0.3	0.92	0.97	1.0

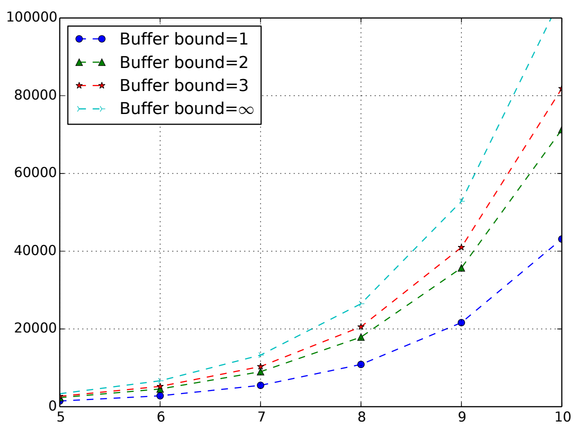


Figure 9. The number of runs (y-axis) explored by DPOR$_{TSO}$ with buffer bounding for different program sizes (x-axis).

show that the search space can be significantly reduced with a small buffer. As we increase the buffer bound, the runtime performance increases as well, gradually reaching that of the standard DPOR$_{TSO}$. Fortunately, from Table 2, we know that even with BB=2 or 3, we can already detect many TSO-related violations.

7. Related Work

The theoretical aspects of verifying concurrent programs under relaxed memory models such as TSO/PSO have been well studied [6, 7, 13]. Specifically, prior works show that reachability in finite-state programs is decidable for TSO and its extension with a write-to-write relaxation. In this work, however, our focus is on improving the practical efficiency of stateless model checking. Toward this end, we have proposed a new dynamic POR algorithm for TSO and PSO.

Dynamic POR was originally proposed by Flanagan and Godefroid [16] for SC. The method is practically appealing because it allows for multithreaded programs written in real languages such as C++, Java and C# to be handled efficiently. Abdulla et al. [1] recently extended DPOR to make it provably optimal and applied it to Erlang. There are also methods for augmenting DPOR with property driven pruning [37] and assertion guided abstraction [21]. However, these existing methods all assume the SC memory model.

Abdulla et al. [2] independently and concurrently proposed a stateless model checking algorithm for TSO/PSO, which shares many similarities with our work. Norris and Demsky [29] also developed a concurrency testing tool called CDSchecker for the C++11 memory model. As we have already mentioned, the main difference between our method and these works is that our method relies on a unified framework for modeling the two different types of nondeterminism under TSO/PSO. That is, we dynamically relax the intra-thread program order by incrementally updating the *enabled* set.

Linden and Wolper [23, 24] also proposed a method for verifying programs under TSO and PSO. However, their method was developed in a different context, i.e., to improve explicit-state model checkers such as SPIN, where the model checker stores the set of visited program states in memory, whereas our goal is to improve stateless model checking, where the model checker never explicitly stores the concrete states.

There are also various heuristic optimization techniques proposed for concurrency testing, but they do not aim to achieve exhaustive coverage. These unsound techniques include, for example, delay bounding, statistical search, context bounding, and synchronization intention [11, 15, 25, 27, 28]. These methods focus primarily on mitigating the nondeterminism in thread scheduling, whereas our work also handles nondeterminism in store buffering.

Beyond stateless model checking, there are also constraint solver based symbolic verification methods. For example, Alglave et al. proposed methods for verifying program under weak memory via program transformation [5] and methods for speeding up bounded model checking [4]. Yang et al. [41, 42] also proposed constraint based methods for checking weak memory models. Furthermore, there is a large body of work on applying POR methods to stateful model checking [31], and SAT-based model checking under SC [20, 34, 35, 38, 39]. These methods are orthogonal to our work.

Finally, hybrid methods have also been proposed in tools such as CheckFence [10], SOBER [9] and RELAXER [12] to detect bugs in concurrent programs under TSO and PSO. They first generate test runs under SC and then amplify these test runs using predictive analysis, to check if any TSO or PSO run inferred from these SC runs is buggy. These TSO/PSO runs share the same transitions as the SC trace, but may have varying delay in store buffers. Hybrid methods in this group differ from our work in that they do not focus on improving the quality of dynamic partial order reduction. Instead, they focus on amplifying existing test runs obtained under SC to increase the chance of detecting TSO and PSO related violations.

8. Conclusions

We have presented a new dynamic partial order reduction method for runtime verification of concurrent programs under TSO and PSO. Our method is sound for checking reachability properties in a concurrent program with a finite and acyclic state space. In addition, we have presented a new method for more aggressively reducing the state space while trying to detect as many TSO/PSO related violations as possible. We have implemented both methods in a runtime verification tool called *rInspect* for multithreaded C/C++ programs. Our experimental results show that the new methods are effective in reducing the search space.

Acknowledgments

This work was primarily supported by the NSF under grants CCF-1149454, CCF-1405697, and CCF-1500024. Partial support was provided by the ONR under grant N00014-13-1-0527. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] P. A. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal dynamic partial order reduction. In *ACM Symposium on Principles of Programming Languages*, 2014.

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas. Stateless model checking for TSO and PSO. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 353–367, 2015.

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

[4] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *European Symposium on Programming*, pages 512–532, 2013.

[5] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*, pages 141–157, 2013.

[6] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *ACM Symposium on Principles of Programming Languages*, pages 7–18, 2010.

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

[8] M. F. Atig, A. Bouajjani, and G. Parlato. Context-bounded analysis of TSO systems. In *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop*, pages 21–38, 2014.

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

[10] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *ACM Conference on Programming Language Design and Implementation*, pages 12–21, 2007.

[11] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

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

[13] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 254–255, 2003.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

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

[16] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Symposium on Principles of Programming Languages*, pages 110–121, 2005. ISBN 1-58113-830-X.

[17] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 3540607617.

[18] P. Godefroid. Model checking for programming languages using VeriSoft. In *ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997. ISBN 0-89791-853-3.

[19] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[20] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[21] M. Kusano and C. Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *International Conference On Automated Software Engineering*, pages 175–186, 2014.

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

[23] A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *International SPIN Workshop on Model Checking Software*, pages 212–226, 2010.

[24] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *International SPIN Workshop on Model Checking Software*, pages 144–160, 2011.

[25] S. Lu, S. Park, and Y. Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Transactions on Parallel and Distributed Systtems*, 23(6):1060–1072, 2012.

[26] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324, 1987. ISBN 0-387-17906-2.

[27] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM Conference on Programming Language Design and Implementation*, pages 446–455, 2007.

[28] S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *ACM Conference on Programming Language Design and Implementation*, pages 543–554, 2012.

[29] B. Norris and B. Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 131–150, 2013.

[30] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, 2009.

[31] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *International Conference on Computer Aided Verification*, pages 377–390, 1994. ISBN 3-540-58179-0.

[32] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 93–107, 2005.

[33] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010. ISSN 0001-0782.

[34] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predicting serializability violations: SMT-based search vs. DPOR-based search. In *Haifa Verification Conference*, 2011.

[35] N. Sinha and C. Wang. On interference abstractions. In *ACM Symposium on Principles of Programming Languages*, pages 423–434, 2011.

[36] SV-COMP. 2014 software verification competition. URL: http://sv-comp.sosy-lab.org/2014/, 2014.

[37] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.

[38] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[39] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 23–32, 2009.

[40] D. L. Weaver and T. Gremond. *The SPARC architecture manual*. PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

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

[42] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: an operational memory model specification framework with integrated model checking capability. *Concurrency - Practice and Experience*, 17(5-6):465–487, 2005.

[43] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *International Workshop on Model Checking Software*, pages 288–305, 2008.