Stateless Model Checking for TSO and PSO

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig,
Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas

Dept. of Information Technology, Uppsala University, Sweden

Abstract. We present a technique for efficient stateless model checking of programs that execute under the relaxed memory models TSO and PSO. The basis for our technique is a novel representation of executions under TSO and PSO, called *chronological traces*. Chronological traces induce a partial order relation on relaxed memory executions, capturing dependencies that are needed to represent the interaction via shared variables. They are optimal in the sense that they only distinguish computations that are inequivalent under the widely-used representation by Shasha and Snir. This allows an optimal dynamic partial order reduction algorithm to explore a minimal number of executions while still guaranteeing full coverage. We apply our techniques to check, under the TSO and PSO memory models, LLVM assembly produced for C/pthreads programs. Our experiments show that our technique reduces the verification effort for relaxed memory models to be almost that for the standard model of sequential consistency. In many cases, our implementation significantly outperforms other comparable tools.

1 Introduction

Verification and testing of concurrent programs is difficult, since one must consider all the different ways in which instructions of different threads can be interleaved. To make matters worse, most architectures implement *relaxed memory models*, such as TSO and PSO [32,4], which make threads interact in even more and subtler ways than by standard interleaving. For example, a processor may reorder loads and stores by the same thread if they target different addresses, or it may buffer stores in a local queue.

A successful technique for finding concurrency bugs (i.e., defects that arise only under some thread schedulings), and for verifying their absence, is *stateless model checking* (SMC) [16], also known as *systematic concurrency testing* [21,35]. Starting from a test, i.e., a way to run a program and obtain some expected result, which is terminating and threadwisely deterministic (e.g. no data-nondeterminism), SMC systematically explores the set of all thread schedulings that are possible during runs of this test. A special runtime scheduler drives the SMC exploration by making decisions on scheduling whenever such decisions may affect the interaction between threads, so that the exploration covers all possible executions and detects any unexpected test results, program crashes, or assertion violations. The technique is completely automatic, has no false positives, does not suffer from memory explosion, and can easily reproduce the concurrency bugs it detects. SMC has been successfully implemented in tools such as VeriSoft [17], CHESS [25], and Concuerror [12].

There are two main problems for using SMC in programs that run under relaxed memory models (RMM). The first problem is that already under the standard model of

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 353–367, 2015.
DOI: 10.1007/978-3-662-46681-0_28

sequential consistency (SC) the number of possible thread schedulings grows exponentially with the length of program execution. This problem has been addressed by *partial order reduction* (POR) techniques that achieve coverage of *all* thread schedulings, by exploring only a representative subset [34,27,15,13]. POR has been adapted to SMC in the form of *Dynamic Partial Order Reduction* (DPOR) [14], which has been further developed in recent years [29,21,19,28,33,2]. DPOR is based on augmenting each execution by a *happens-before relation*, which is a partial order that captures dependencies between operations of the threads. Two executions can be regarded as equivalent if they induce the same happens-before relation, and it is therefore sufficient to explore one execution in each equivalence class (called a *Mazurkiewicz trace* [24]). DPOR algorithms guarantee to explore at least one execution in each equivalence class, thus attaining full coverage with reduced cost. A recent optimal algorithm [2] guarantees to explore *exactly* one execution per equivalence class.

The second problem is that in order to extend SMC to handle relaxed memory models, the operational semantics of programs must be extended to represent the effects of RMM. The natural approach is to augment the program state with additional structures, e.g., store buffers in the case of TSO, that model the effects of RMM [3,5,26]. This causes blow-ups in the number of possible executions, in addition to those possible under SC. However, most of these additional executions are equivalent to some SC execution. To efficiently apply SMC to handle RMM, we must therefore extend DPOR to avoid redundant exploration of equivalent executions. The natural definition of "equivalent" under RMM can be derived from the abstract representation of executions due to Shasha and Snir [31], here called *Shasha-Snir traces*, which is often used in model checking and runtime verification [18,20,10,11,7,8]. Shasha-Snir traces consist of an ordering relation between dependent operations, which generalizes the standard happens-before relation on SC executions; indeed, under SC, the equivalence relation induced by Shasha-Snir traces coincides with Mazurkiewicz traces. It would thus be natural to base DPOR for RMM on the happens-before relation induced by Shasha-Snir traces. However, this relation is in general cyclic (due to reorderings possible under RMM) and can therefore not be used as a basis for DPOR (since it is not a partial order). To develop an efficient technique for SMC under RMM we therefore need to find a different representation of executions under RMM. The representation should define an acyclic happens-before relation. Also, the induced trace equivalence should coincide with the equivalence induced by Shasha-Snir traces.

Contribution. In this paper, we show how to apply SMC to TSO and PSO in a way that achieves maximal possible reduction using DPOR, in the sense that redundant exploration of equivalent executions is avoided. A cornerstone in our contribution is a novel representation of executions under RMM, called *chronological traces*, which define a happens-before relation on the events in a carefully designed representation of program executions. Chronological traces are a succinct canonical representation of executions, in the sense that there is a one-to-one correspondence between chronological traces and Shasha-Snir traces. Furthermore, the happens-before relation induced by chronological traces is a partial order, and can therefore be used as a basis for DPOR. In particular, the Optimal-DPOR algorithm of [2] will explore exactly one execution per Shasha-Snir trace. In particular, for so-called *robust* programs that are not affected

by RMM (these include data-race-free programs), Optimal-DPOR will explore as many executions under RMM as under SC: this follows from the one-to-one correspondence between chronological traces and Mazurkiewicz traces under SC. Furthermore, robustness can itself be considered a correctness criterion, which can also be automatically checked with our method (by checking whether the number of equivalence classes is increased when going from SC to RMM).

We show the power of our technique by using it to implement an efficient stateless model checker, which for C programs with pthreads explores all executions of a test-case or a program, up to some bounded length. During exploration of an execution, our implementation generates the corresponding chronological trace. Our implementation employs the source-DPOR algorithm [2], which is simpler than Optimal-DPOR, but about equally effective. Our experimental results for analyses under SC, TSO and PSO of number of intensely racy benchmarks and programs written in C/pthreads, shows that (i) the effort for verification under TSO and PSO is not much larger than the effort for verification under SC, and (ii) our implementation compares favourably against CBMC, a state-of-the-art bounded model checking tool, showing the potential of our approach.

2 Overview of Main Concepts

This section informally motivates and explains the main concepts of the paper. To focus the presentation, we consider mainly the TSO model. TSO is relevant because it is implemented in the widely used x86 as well as SPARC architectures. We first introduce TSO and its semantics. Thereafter we introduce Shasha-Snir traces, which abstractly represent the orderings between dependent events in an execution. Since Shasha-Snir traces are cyclic, we introduce an extended representation of executions, for which a natural happens-before relation is acyclic. We then describe how this happens-before relation introduces undesirable distinctions between executions, and how our new representation of chronological traces remove these distinctions. Finally, we illustrate how a DPOR algorithm exploits the happens-before relation induced by chronological traces to explore only a minimal number of executions, while still guaranteeing full coverage.

p	q
store: $\mathbf{x}$:=1	store: $\mathbf{y}$:=1
load: \$$r$:=$\mathbf{y}$	load: \$$s$:=$\mathbf{x}$

Fig. 1. A program implementing the classic idiom of Dekker's mutual exclusion algorithm

TSO — an Introduction. TSO relaxes the ordering between stores and subsequent loads to different memory locations. This can be explained operationally by equipping each thread with a *store buffer* [30], which is a FIFO queue that contains pending store operations. When a thread executes a store instruction, the store does not immediately affect memory. Instead it is delayed and enqueued in the store buffer. Nondeterministically, at some later point an *update* event occurs, dequeueing the oldest store from the store buffer and updating the memory correspondingly. Load instructions take effect immediately, without being delayed. Usually a load reads a value from memory. However, if the store buffer of the same thread contains a store to the same memory location, the value is instead taken from the store in the store buffer.

To see why this buffering semantics may cause unexpected program behaviors, consider the small program in Fig. 1. It consists of two threads p and q. The thread p first stores 1 to the memory location **x**, and then loads the value at memory location **y** into its register $\$r$. The thread q is similar. All memory locations and registers are assumed to have initial values 0.

```
p: store: x :=1  // Enqueue store
p: load: $r:=y   // Load value 0
    q: store: y:=1   // Enqueue store
    q: update        // y = 1 in memory
    q: load: $s:=x   // Load value 0
p: update        // x = 1 in memory
```

Fig. 2. An execution of the program in Fig. 1. Notice that $\$r = \$s = 0$ at the end.

It is easy to see that under the SC semantics, it is impossible for the program to terminate in a state where both registers $\$r$ and $\$s$ hold the value 0. However, under the buffering semantics of TSO, such a final state is possible. Fig. 2 shows one such program execution. We see that the store to **x** happens at the beginning of the execution, but does not take effect with respect to memory until the very end of the execution. Thus the store to **x** and the load to **y** appear to take effect in an order opposite to how they occur in the program code. This allows the execution to terminate with $\$r = \$s = 0$.

Shasha-Snir Traces for TSO. Partial order reduction is based on the idea of capturing the possible orderings between dependent operations of different threads by means of a happens-before relation. When threads interact via shared variables, two instructions are considered dependent if they access the same global variable, and at least one is a write. For relaxed memory models, Shasha and Snir [31] introduced an abstract representation of executions, here referred to as *Shasha-Snir traces*, which captures such dependencies in a natural way. Shasha-Snir traces induce equivalence classes of executions. Under sequential consistency, those classes coincide with the Mazurkiewicz traces. Under a relaxed memory model, there are also additional Shasha-Snir traces corresponding to the non-sequentially consistent executions.

A Shasha-Snir trace is a directed graph, where edges capture observed event orderings. The nodes in a Shasha-Snir trace are the executed instructions. For each thread, there are edges between each pair of subsequent instructions, creating a total order for each thread. For two instructions i and j in different threads, there is an edge $i \to j$ in a trace when i causally precedes j. This happens when j reads a value that was written by i, when i reads a memory location that is subsequently updated by j, or when i and j are subsequent writes to the same memory location. In Fig. 3 we show the Shasha-Snir trace for the execution in Fig. 2.

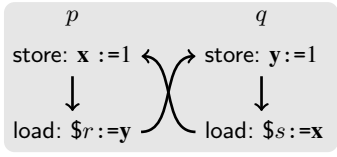


Fig. 3. The Shasha-Snir trace corresponding to the execution in Fig. 2

Making the Happens-Before Relation Acyclic. Shasha-Snir traces naturally represent the dependencies between operations in an execution, and are therefore a natural basis for applying DPOR. However, a major problem is that the happens-before relation induced by the edges is in general cyclic, and thus not a partial order. This can be seen already in the graph in Fig. 3. This problem can be addressed by adding nodes that represent explicit update events. That would be natural since such events occur in the

representation of the execution in Fig. 2. When we consider the edges of the Shasha-Snir trace, we observe that although there is a conflict between p : load: $\$r$:=**y** and q : store: **y**:=1, swapping their order in the execution in Fig. 2 has no observable effect; the load still gets the same value from memory. Therefore, we should only be concerned with the order of the load relative to the update event q : update.

These observations suggest to define a representation of traces that separates stores from updates. In Fig. 4 we have redrawn the trace from Fig. 3. Updates are separated from stores, and we order updates, rather than stores, with operations of other threads. Thus, there are edges between updates to and loads from the same memory location, and between two updates to the same memory location. In Fig. 4, there is an edge from each store to the corresponding update, reflecting the principle that the update cannot occur before the store. There are edges between loads and updates of the same memory location, reflecting that swapping their order will affect the observed values. However, notice that for this program there are no edges between the updates and loads of the same thread, since they access different memory locations.

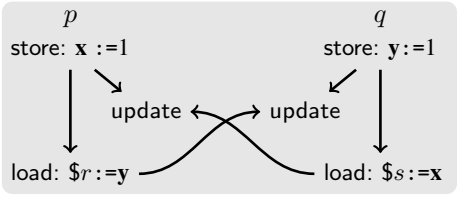


Fig. 4. A trace for the execution in Fig. 2 where updates are separated from stores

Chronological Traces for TSO. Although the new representation is a valid partial order, it will in many cases distinguish executions that are semantically equivalent according to the Shasha-Snir traces. The reason for this is TSO buffer forwarding: When a thread executes a load to a memory location **x**, it will first check its store buffer. If the buffer contains a store to **x**, then the load returns the value of the newest such store buffer entry instead of loading the value from memory. This causes problems for a happens-before relation that orders all updates and loads of the same memory location.

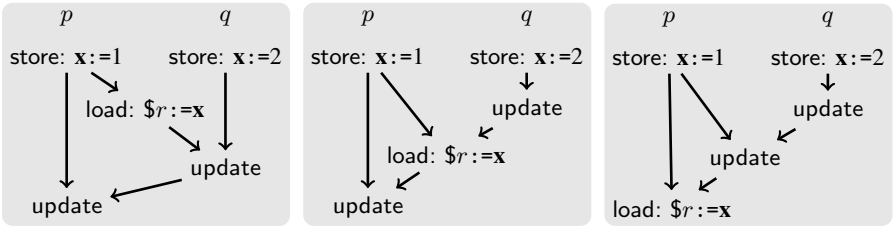
For example, consider the program shown in Fig. 5. Any execution of this program will have two updates and one load to **x**. Those accesses can be permuted in six different ways. Fig. 6(a), 6(b) and 6(c) show three of the corresponding happens-before relations. In Fig. 6(a) and 6(b) the load is satisfied by buffer forwarding, and in 6(c) by a read from memory. These three relations all correspond to the same Shasha-Snir trace, shown in Fig. 7(a), and they all have the same observable behavior, since the value of the load is obtained from the same store. Hence, we should find a representation of executions that does not distinguish between these three cases.

p	q
store: **x**:=1	store: **x**:=2
load: $\$r$:=**x**	

Fig. 5. A program illustrating buffer forwarding

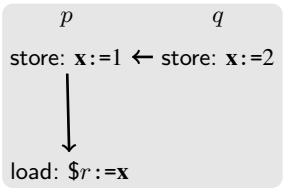
We can now describe *chronological traces*, our representation which solves the above problems, by omitting some of the edges, leaving some nodes unrelated. More precisely, edges between loads and updates should be omitted in the following cases.

1. A load is never related to an update originating in the same thread. This captures the intuition that swapping the order of such a load and update has no effect other

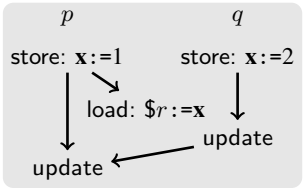


(a) (b) (c)

Fig. 6. Three redundant happens-before relations for Fig. 5



(a) A Shasha-Snir trace corresponding to all three traces of Fig. 6



(b) The three traces can be merged into this single trace

Fig. 7. Traces unifying the ones in Fig. 6

than changing a load from memory into a load of the same value from buffer, as seen when comparing Fig. 6(b) and 6(c).

2. A load ld from a memory location **x** by a thread p is never related to an update by an another thread q, if the update by q precedes some update to **x** originating in a store by p that precedes ld. This is because the value written by the update of q is effectively hidden to the load ld by the update to **x** by p. Thus, when we compare Fig. 6(a) and 6(b), we see that the order between the update by q and the load is irrelevant, since the update by q is hidden by the update by p (note that the update by p originates in a store that precedes the load).

When we apply these rules to the example of Fig. 5, all of the three representations in Fig. 6(a), 6(b), and 6(c) merge into a single representation shown in Fig. 7(b). In total, we reduce the number of distinguished cases for the program from six to three. This is indeed the minimal number of cases that must be distinguished by any representation, since the different cases result in different values being loaded by the load instruction or different values in memory at the end of the execution. Our proposed representation is optimal for the programs in Fig. 1 and 5. In Theorem 1 of Section 3 we will show that such an optimality result holds in general.

Chronological Traces for PSO. The TSO and PSO memory models are very similar. Adapting our techniques to PSO is done by slightly altering the definition of chronological traces. The details can be found in our technical report [1].

DPOR Based on Chronological Traces. Here, we illustrate how stateless model checking performs DPOR based on chronological traces, in order to explore one execution per chronological trace. As example, we use the small program of Fig. 5.

The algorithm initially explores an arbitrary execution of the program, and simultaneously generates the corresponding chronological trace. In our example, this execution can be the one shown in Fig. 8(a), along with its chronological trace. The algorithm then finds those edges of the chronological trace that can be reversed by changing the thread scheduling of the execution. In Fig. 8(a), the reversible edges are the ones from p : update to q : update, and from p : load: $\$r$:= **x** to q : update. For each such edge, the program is executed with this edge reversed. Reversing an edge can potentially lead to a completely different continuation of the execution, which must then be explored.

In the example, reversing the edge from p : load: $\$r$:= **x** to q : update will generate the execution and chronological trace in Fig. 8(b). Notice that the new execution is observably different from the previous one: the load reads the value 2 instead of 1.

The chronological traces in both Fig. 8(a) and 8(b) display a reversible edge from p : update to q : update. The algorithm therefore initiates an execution where q : update is performed before p : update. The algorithm will generate the execution and chronological trace in Fig. 8(c).

Notice that the only reversible edge in Fig. 8(c) is the one from q : update to p : update. However, executing p : update before q : update has already been explored in Fig. 8(a) and Fig. 8(b). Since there are no more edges that can be reversed, SMC terminates, having examined precisely the three chronological traces that exist for the program of Fig. 5.

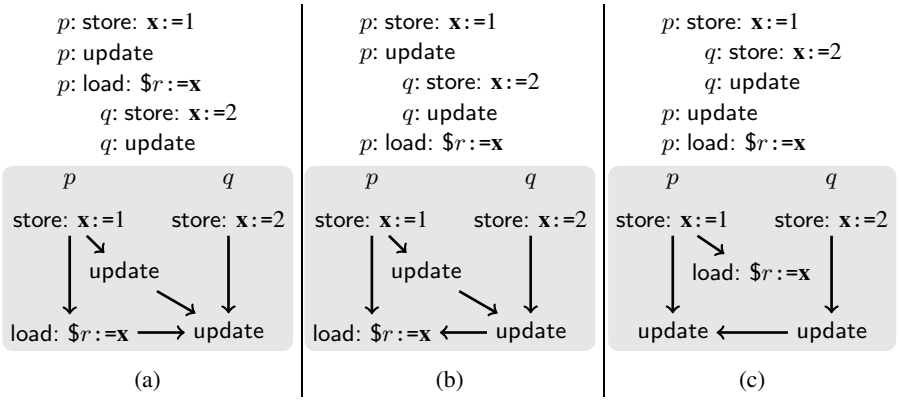


Fig. 8. How SMC with DPOR explores the program of Fig. 5

3 Formalization

In this section we summarize our formalization of the concepts of Section 2. We introduce our representation of program executions, define chronological traces, formalize Shasha-Snir traces for TSO, and prove a one-to-one correspondence between chronological traces and Shasha-Snir traces. The formalization is self-contained, but for lack of space, we sometimes use precise English rather than formal notation. A more fully formalized version, and theorem proofs, can be found in our technical report [1].

Parallel Programs. We consider parallel programs consisting of a number of threads that run in parallel, each executing a deterministic code, written in an assembly-like programming language. The language includes instructions store: $\mathbf{x}$:=$\$r$, load: $\$r$:=$\mathbf{x}$, and fence. Other instructions do not access memory, and their precise syntax and semantics are ignored for brevity. Here, and in the remainder of this text, **x**, **y**, **z** are used to name memory locations, u, v, w are used to name values, and $\$r$, $\$s$, $\$t$ are used to name processor registers. We use TID to denote the set of all thread identifiers.

Formal TSO Semantics. We formalize the TSO model by an operational semantics. Define a *configuration* as a pair $(\mathbb{L}, \mathbb{M})$, where $\mathbb{M}$ maps memory locations to values, and $\mathbb{L}$ maps each thread p to a local configuration of the form $\mathbb{L}(p) = (\mathbb{R}, \mathbb{B})$, where $\mathbb{R}$ is the state of local registers and program counter of p, and $\mathbb{B}$ is the contents of the store buffer of p. This content is a word over pairs $(\mathbf{x}, v)$ of memory locations and values. We let the notation $\mathbb{B}(\mathbf{x})$ denote the value v such that $(\mathbf{x}, v)$ is the rightmost pair in $\mathbb{B}$ of form $(\mathbf{x}, \_)$. If there is no such pair in $\mathbb{B}$, then $\mathbb{B}(\mathbf{x}) = \perp$.

In order to accommodate memory updates in our operational semantics, we assume that for each thread $p \in \mathsf{TID}$, there is an auxiliary thread $\mathsf{upd}(p)$, which nondeterministically performs memory updates from the store buffer of p. We use $\mathsf{AuxTID} = \{\mathsf{upd}(p) | p \in \mathsf{TID}\}$ to denote the set of auxiliary thread identifiers. We use p and q to refer to real or auxiliary threads in $\mathsf{TID} \cup \mathsf{AuxTID}$ as convenient.

For configurations $c = (\mathbb{L}, \mathbb{M})$ and $c' = (\mathbb{L}', \mathbb{M}')$, we write $c \xrightarrow{p} c'$ to denote that from configuration c, thread p can execute its next instruction, thereby changing the configuration into c'. Let $\mathbb{L}(p) = (\mathbb{R}, \mathbb{B})$, and $\mathbb{R}_{\mathsf{pc}}$ be obtained from $\mathbb{R}$ by advancing the program counter after p executes its next instruction. Depending on this next instruction *op*, we have the following cases.

Store: If *op* has the form store: $\mathbf{x}$:=$\$r$, then $c \xrightarrow{p} c'$ iff $\mathbb{M}' = \mathbb{M}$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}, \mathbb{B} \cdot (\mathbf{x}, v))]$ where $v = \mathbb{R}(\$r)$, i.e., instead of updating the memory, we insert the entry $(\mathbf{x}, v)$ at the end of the store buffer of the thread.

Load: If *op* has the form load: $\$r$:=$\mathbf{x}$, then $\mathbb{M}' = \mathbb{M}$ and either

1. (**From memory**) $\mathbb{B}(\mathbf{x}) = \perp$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}[\$r \hookleftarrow \mathbb{M}(\mathbf{x})], \mathbb{B})]$, i.e., there is no entry for **x** in the thread's own store buffer, so the value is read from memory, or
2. (**Buffer forwarding**) $\mathbb{B}(\mathbf{x}) \neq \perp$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}[\$r \hookleftarrow \mathbb{B}(\mathbf{x})], \mathbb{B})]$, i.e., p reads the value of **x** from its *latest* entry in its store buffer.

Fence: If *op* has the form fence, then $c \xrightarrow{p} c'$ iff $\mathbb{B} = \varepsilon$ and $\mathbb{M}' = \mathbb{M}$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}, \mathbb{B})]$. A fence can only be executed when the store buffer of the thread is empty.

Update: In addition to instructions which are executed by the threads, at any point when a store buffer is non-empty, an *update* event may nondeterministically occur. The memory is then updated according to the oldest (leftmost) letter in the store buffer, and that letter is removed from the buffer. To formalize this, we will assume that the auxiliary thread $\mathsf{upd}(p)$ executes a pseudo-instruction $\mathsf{u}(\mathbf{x})$. We then say that $c \xrightarrow{\mathsf{upd}(p)} c'$ iff $\mathbb{B} = (\mathbf{x}, v) \cdot \mathbb{B}'$ for some **x**, v, $\mathbb{B}'$ and $\mathbb{M}' = \mathbb{M}[\mathbf{x} \hookleftarrow v]$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}, \mathbb{B}')]$.

Program Executions. A program execution is a sequence $c_0 \xrightarrow{p_1} c_1 \xrightarrow{p_2} \cdots \xrightarrow{p_n} c_n$ of configurations related by transitions labelled by actual or auxiliary thread IDs. Since each transition of each program thread (including the auxiliary threads of form $\mathsf{upd}(q)$) is deterministic, a program run is uniquely determined by its sequence of thread IDs. We will therefore define an *execution* as a word of *events*. Each event represents a transition in the execution as a triple (p, i, j), where p is a regular or auxiliary thread executing an instruction i (which can possibly be an update), and the natural number j is such that the event is the jth event of p in the execution.

Chronological Traces. We can now introduce the main conceptual contribution of the paper, viz. *chronological traces*. For an execution τ we define its chronological trace $\mathcal{T}_C(\tau)$ as a directed graph $\langle V, E\rangle$. The vertices V are all the events in τ (both events representing instructions and events representing updates). The edges are the union of six relations: $E = \rightarrow_\tau^{\mathsf{po}} \cup \rightarrow_\tau^{\mathsf{su}} \cup \rightarrow_\tau^{\mathsf{uu}} \cup \rightarrow_\tau^{\mathsf{src\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{cf\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{uf}}$. These edge relations are defined as follows, for two arbitrary events $e = (p, i, j), e' = (p', i', j') \in V$:

Program Order: $e \rightarrow_\tau^{\mathsf{po}} e'$ iff $p = p'$ and $j' = j + 1$, i.e., e and e' are consecutive events of the same thread.

Store to Update: $e \rightarrow_\tau^{\mathsf{su}} e'$ iff e' is the update event corresponding to the store e.

Update to Update: $e \rightarrow_\tau^{\mathsf{uu}} e'$ iff $i = \mathsf{u}(\mathbf{x})$ and $i' = \mathsf{u}(\mathbf{x})$ for some $\mathbf{x}$, and e and e' are consecutive updates to the memory location $\mathbf{x}$.

Source: $e \rightarrow_\tau^{\mathsf{src\text{-}ct}} e'$ iff e' is a load which reads the value of the update event e, which is from a different process. Notice that this definition excludes the possibility of $p = \mathsf{upd}(p')$; a load is never src-related to an update from the same thread.

Conflict: $e \rightarrow_\tau^{\mathsf{cf\text{-}ct}} e'$ iff e' is the update that overwrites the value read by e.

Update to Fence: $e \rightarrow_\tau^{\mathsf{uf}} e'$ iff $i = \mathsf{u}(\mathbf{x})$ for some $\mathbf{x}$, and $i' = \mathsf{fence}$ and $p = \mathsf{upd}(p')$ and e is the latest update by p which occurs before e' in τ. The intuition here is that the fence cannot be executed until all pending updates of the same thread have been flushed from the buffer. Hence the updates are ordered before the fence, and the chronological trace has an edge from the last of these updates to the fence event.

Shasha-Snir Traces. We will now formalize Shasha-Snir traces, and prove that chronological traces are equivalent to Shasha-Snir traces, in the sense that they induce the same equivalence relation on executions. We first recall the definition of Shasha-Snir traces. We follow the formalization by Bouajjani *et al.* [8].

First, we introduce the notion of a completed execution: An execution τ is *completed* when all stores have reached memory by means of a corresponding update event. In the context of Shasha-Snir traces, we will restrict ourselves to completed executions.

For a completed execution τ, we define the Shasha-Snir trace of τ as the graph $\mathcal{T}(\tau) = \langle V, E\rangle$ where V is the set of all non-update events (p, i, j) in τ (i.e., $i \neq \mathsf{u}(\mathbf{x})$ for all $\mathbf{x}$). The edges E is the union of four relations $E = \rightarrow_\tau^{\mathsf{po}} \cup \rightarrow_\tau^{\mathsf{st}} \cup \rightarrow_\tau^{\mathsf{src\text{-}ss}} \cup \rightarrow_\tau^{\mathsf{cf\text{-}ss}}$, where $\rightarrow_\tau^{\mathsf{po}}$ (program order) is the same as for Chronological traces, and where, letting $e = (p, i, j)$ and $e' = (p', i', j')$:

Store Order: $e \rightarrow_\tau^{\mathsf{st}} e'$ iff i and i' are two stores, whose corresponding updates are consecutive updates to the same memory location. I.e., store order defines a total order on all the stores to each memory location, based on the order in which they reach memory.

Source: $e \rightarrow_\tau^{\mathsf{src\text{-}ss}} e'$ iff e' is a load which reads its value from e, via memory or by buffer forwarding.

Conflict: $e \rightarrow_\tau^{\mathsf{cf\text{-}ss}} e'$ iff e' is the store which overwrites the value read by e.

We are now ready to state the equivalence theorem.

Theorem 1. (Equivalence of Shasha-Snir Traces and Chronological Traces) *For a given program $\mathcal{P}$ with two completed executions τ, τ', it holds that $\mathcal{T}(\tau) = \mathcal{T}(\tau')$ iff $\mathcal{T}_C(\tau) = \mathcal{T}_C(\tau')$.*

DPOR for TSO. A DPOR algorithm can exploit Chronological traces to perform stateless model checking of programs that execute under TSO (and PSO), as illustrated at the end of Section 2. The explored executions follow the semantics of TSO in Section 3. For each execution, its happens-before relation, which is the transitive closure of the edge relation $E = \rightarrow_\tau^{\mathsf{po}} \cup \rightarrow_\tau^{\mathsf{su}} \cup \rightarrow_\tau^{\mathsf{uu}} \cup \rightarrow_\tau^{\mathsf{src\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{cf\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{uf}}$ of the corresponding chronological trace, is computed on the fly. This happens-before relation can in principle be exploited by any DPOR algorithm to explore at least one execution per equivalence class induced by Shasha-Snir traces. We state the following theorem of correctness.

Theorem 2. (Correctness of DPOR Algorithms) *The algorithms* Source-DPOR *and* Optimal-DPOR *of [2], based on the happens-before relation induced by chronological traces, explore at least one execution per equivalence class induced by Shasha-Snir traces. Moreover,* Optimal-DPOR *explores exactly one execution per equivalence class.*

4 Implementation

To show the effectiveness of our techniques we have implemented a stateless model checker for C programs. The tool, called Nidhugg, is available as open source at `https://github.com/nidhugg/nidhugg`. Major design decisions have been that Nidhugg: (i) should not be bound to a specific hardware architecture and (ii) should use an existing, mature implementation of C semantics, not implement its own. Our choice was to use the LLVM compiler infrastructure [23] and work at the level of its intermediate representation (IR). LLVM IR is low-level and allows us to analyze assembly-like but target-independent code which is produced after employing all optimizations and transformations that the LLVM compiler performs till this stage.

Nidhugg detects assertion violations and robustness violations that occur under the selected memory model. We implement the Source-DPOR algorithm from Abdulla *et al.* [2], adapted to relaxed memory in the manner described in this paper. Before applying Source-DPOR, each spin loop is replaced by an equivalent single load and assume statement. This substantially improves the performance of Source-DPOR, since a waiting spin loop may generate a huge number of improductive loads, all returning

the same wrong value; all of these loads will cause races, which will cause the number of explored traces to explode. Exploration of program executions is performed by interpretation of LLVM IR, based on the interpreter lli which is distributed with LLVM. We support concurrency through the pthreads library. This is done by hooking calls to pthread functions, and executing changes to the execution stacks (adding new threads, joining, etc.) as appropriate within the interpreter.

5 Experimental Results

We have applied our implementation to several intensely racy benchmarks, all implemented in C/pthreads. They include classical benchmarks, such as Dekker's, Lamport's (fast) and Peterson's mutual exclusion algorithms. Others, such as indexer.c, are designed to showcase races that are hard to identify statically. Yet others (stack_safe.c) use pthread mutexes to entirely avoid races. Lamport's algorithm and stack_safe.c originate from the TACAS Competition on Software Verification (SV-COMP). Some benchmarks originate from industrial code: apr_1.c, apr_2.c, pgsql.c and parker.c.

We show the results of our tool Nidhugg in Table 1. For comparison we also include the results of two other analysis tools, CBMC [6] and goto-instrument [5], which also target C programs under relaxed memory. The techniques of goto-instrument and CBMC are described in more detail in Section 6.

All experiments were run on a machine equipped with a 3 GHz Intel i7 processor and 6 GB RAM running 64-bit Linux. We use version 4.9 of goto-instrument and CBMC. The benchmarks have been tweaked to work for all tools, in communication with the developers of CBMC and goto-instrument. All benchmarks are available at `https://github.com/nidhugg/benchmarks_tacas2015`.

Table 1 shows that our technique performs well compared to the other tools for most of the examples. We will briefly highlight a few interesting results.

We see that in most cases Nidhugg pays a very modest performance price when going from sequential consistency to TSO and PSO. The explanation is that the number of executions explored by our stateless model checker is close to the number of Shasha-Snir traces, which increases very modestly when going from sequential consistency to TSO and PSO for typical benchmarks. Consider for example the benchmark stack_safe.c, which is robust, and therefore has equally many Shasha-Snir traces (and hence also chronological traces) under all three memory models. Our technique is able to benefit from this, and has almost the same run time under TSO and PSO as under SC.

We also see that our implementation compares favourably against CBMC, a state-of-the-art bounded model checking tool, and goto-instrument. For several benchmarks, our implementation is several orders of magnitude faster.

The effect of the optimization to replace each spin loop by a load and assume statement can be seen in the pgsql.c benchmark. For comparison, we also include the benchmark pgsql_bnd.c, where the spin loop has been modified such that Nidhugg fails to automatically replace it by an assume statement.

The only other benchmark where Nidhugg is not faster is fib_true.c. The benchmark has two threads that perform the actual work, and one separate thread that checks the correctness of the computed value, causing many races, as in the case of spin loops.

Table 1. Analysis times (in seconds) for our implementation Nidhugg, as well as CBMC and goto-instrument under the SC, TSO and PSO memory models. Stars (*) indicate that the analysis discovered an error in the benchmark. A t/o entry means that the tool did not terminate within 10 minutes. An ! entry means that the tool crashed. ~~Struck-out~~ entries mean that the tool gave the wrong result. In the fence column, a dash (-) means that no fences have been added to the benchmark, a memory model indicates that fences have been (manually) added to make the benchmark correct under that and stronger memory models. The LB column shows the loop unrolling depth. Superior run times are shown in bold face.

			CBMC			goto-instrument			Nidhugg		
	fence	LB	SC	TSO	PSO	SC	TSO	PSO	SC	TSO	PSO
apr_1.c	-	5	t/o	t/o	t/o	t/o	!	!	**5.88**	**6.06**	**16.98**
apr_2.c	-	5	t/o	t/o	t/o	!	!	!	**2.60**	**2.20**	**5.39**
dcl_singleton.c	-	7	5.95	31.47	*18.01	5.33	5.36	*0.18	**0.08**	**0.08**	***0.08**
dcl_singleton.c	pso	7	5.88	30.98	29.45	5.20	5.18	5.17	**0.08**	**0.08**	**0.08**
dekker.c	-	10	2.42	*3.17	*2.84	1.68	*4.00	*220.11	**0.10**	***0.11**	***0.09**
dekker.c	tso	10	2.39	5.65	*3.51	1.62	297.62	t/o	**0.11**	**0.12**	***0.08**
dekker.c	pso	10	2.55	5.31	4.83	1.72	428.86	t/o	**0.11**	**0.12**	**0.12**
fib_false.c	-	-	*1.63	*3.38	*3.00	***1.60**	***1.58**	***1.56**	*2.39	*5.57	*6.20
fib_false_join.c	-	-	*0.98	*1.10	*1.91	*1.31	*0.88	*0.80	***0.32**	***0.62**	***0.71**
fib_true.c	-	-	**6.28**	9.39	7.72	6.32	**7.63**	**7.62**	25.83	75.06	86.32
fib_true_join.c	-	-	6.61	8.37	10.81	7.09	5.94	5.92	**1.20**	**2.88**	**3.19**
indexer.c	-	5	193.01	210.42	214.03	191.88	70.42	69.38	**0.10**	**0.09**	**0.09**
lamport.c	-	8	7.78	*11.63	*10.53	6.89	t/o	t/o	**0.08**	***0.08**	***0.08**
lamport.c	tso	8	7.60	26.31	*15.85	6.80	513.67	t/o	**0.09**	**0.08**	***0.07**
lamport.c	pso	8	7.72	30.92	27.51	7.43	t/o	t/o	**0.08**	**0.08**	**0.08**
parker.c	-	10	12.34	*91.99	*86.10	11.63	~~9.70~~	~~9.65~~	**1.50**	***0.09**	***0.08**
parker.c	pso	10	12.72	141.24	166.75	11.76	10.66	10.64	**1.50**	**1.92**	**2.94**
peterson.c	-	-	0.35	*0.38	*0.35	0.18	*0.20	*0.21	**0.07**	***0.07**	***0.07**
peterson.c	tso	-	0.35	0.39	*0.35	0.19	0.18	~~0.56~~	**0.07**	**0.07**	***0.07**
peterson.c	pso	-	0.35	0.41	0.40	0.18	0.18	0.19	**0.07**	**0.07**	**0.08**
pgsql.c	-	8	19.80	60.66	*4.63	21.03	46.57	*296.77	**0.08**	**0.07**	***0.08**
pgsql.c	pso	8	23.93	71.15	121.51	19.04	t/o	t/o	**0.07**	**0.07**	**0.08**
pgsql_bnd.c	pso	(4)	**3.57**	**9.55**	**12.68**	3.59	t/o	t/o	89.44	106.04	112.60
stack_safe.c	-	-	44.53	516.01	496.36	45.11	42.39	42.50	**0.34**	**0.36**	**0.43**
stack_unsafe.c	-	-	*1.40	*1.87	*2.08	*1.00	*0.81	*0.79	***0.08**	***0.08**	***0.09**
szymanski.c	-	-	0.40	*0.44	*0.43	0.23	*0.89	*1.16	**0.07**	***0.13**	***0.07**
szymanski.c	tso	-	0.40	0.50	*0.43	0.23	0.23	~~2.48~~	**0.08**	**0.08**	***0.07**
szymanski.c	pso	-	0.39	0.50	0.49	0.23	0.24	0.24	**0.08**	**0.08**	**0.08**

We show with the benchmark fib_true_join.c that in this case, the problem can be alleviated by forcing the threads to join before checking the result.

Most benchmarks in Table 1 are small program cores, ranging from 36 to 118 lines of C code, exhibiting complicated synchronization patterns. To show that our technique is also applicable to real life code, we include the benchmarks apr_1.c and apr_2.c. They each contain approximately 8000 lines of code taken from the Apache Portable Runtime library, and exercise the library primitives for thread management, locking, and memory pools. Nidhugg is able to analyze the code within a few seconds. We notice that despite the benchmarks being robust, the analysis under PSO suffers a slowdown of about three times compared to TSO. This is because the benchmarks access a large number of different memory locations. Since PSO semantics require one store buffer per memory location, this affects analysis under PSO more than under SC and TSO.

6 Related Work

To the best of our knowledge, our work is the first to apply stateless model checking techniques to the setting of relaxed memory models; see e.g. [2] for a recent survey of related work on stateless model checking and dynamic partial order reduction techniques. There have been many works dedicated to the verification and checking of programs running under RMM (e.g., [18,20,22,3,10,11,7,8,9,36]). Some of them propose *precise* analyses for checking safety properties or robustness of finite-state programs under TSO (e.g., [3,8]). Others describe monitoring and testing techniques for programs under RMM (e.g., [10,11,22]). There are also a number of efforts to design bounded model checking techniques for programs under RMM (e.g., [36,9]) which encode the verification problem in SAT.

The two closest works to ours are those presented in [6,5]. The first of them [6] develops a bounded model checking technique that can be applied to different memory models (e.g., TSO, PSO, and Power). That technique makes use of the fact that the trace of a program under RMM can be viewed as a partially ordered set. This results in a bounded model checking technique aware of the underlying memory model when constructing the SMT/SAT formula. The second line of work reduces the verification problem of a program under RMM to verification under SC of a program constructed by a code transformation [5]. This technique tries to encode the effect of the RMM semantics by augmenting the input program with buffers and queues. This work introduces also the notion of Xtop objects. Although an Xtop object is a valid acyclic representation of Shasha-Snir traces, it will in many cases distinguish executions that are semantically equivalent according to the Shasha-Snir traces. This is never the case for chronological traces. An extensive experimental comparison with the corresponding tools [6,5] under the TSO and PSO memory models was given in Section 5.

7 Concluding Remarks

We have presented the first technique for efficient *stateless model checking* which is aware of the underlying relaxed memory model. To this end we have introduced *chronological traces* which are novel representations of executions under the TSO and PSO

memory models, and induce a happens-before relation that is a partial order and can be used as a basis for DPOR. Furthermore, we have established a strict one-to-one correspondence between chronological and Shasha-Snir traces. Nidhugg, our publicly available tool, detects bugs in LLVM assembly code produced for C/pthreads programs and can be instantiated to the SC, TSO, and PSO memory models. We have applied Nidhugg to several programs, both benchmarks and of considerable size, and our experimental results show that our technique offers significantly better performance than both CBMC and goto-instrument in many cases.

We plan to extend Nidhugg to more memory models such as Power, ARM, and the C/C++ memory model. This will require to adapt the definition chronological traces to them in order to also guarantee the one-to-one correspondence with Shasha-Snir traces.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO (2015) arXiv:1501.02069
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL, pp. 373–384. ACM (2014)
3. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
4. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (1996)
5. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
6. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
7. Alglave, J., Maranget, L.: Stability in weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011)
8. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)
9. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21. ACM (2007)
10. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
11. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 11–25. Springer, Heidelberg (2011)
12. Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in Erlang programs. In: ICST, pp. 154–163. IEEE (2013)
13. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. STTT 2(3), 279–287 (1999)
14. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121. ACM (2005)
15. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)

16. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL, pp. 174–186. ACM (1997)
17. Godefroid, P.: Software model checking: The VeriSoft approach. Formal Methods in System Design 26(2), 77–101 (2005)
18. Krishnamurthy, A., Yelick, K.A.: Analyses and optimizations for shared address space programs. J. Parallel Distrib. Comput. 38(2), 130–144 (1996)
19. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 308–322. Springer, Heidelberg (2010)
20. Lee, J., Padua, D.A.: Hiding relaxed memory consistency with a compiler. IEEE Trans. Computers 50(8), 824–833 (2001)
21. Lei, Y., Carver, R.: Reachability testing of concurrent programs. IEEE Trans. Softw. Eng. 32(6), 382–403 (2006)
22. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI, pp. 429–440. ACM (2012)
23. The LLVM compiler infrastructure, `http://llvm.org`
24. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, Springer, Heidelberg (1987)
25. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI, pp. 267–280. USENIX (2008)
26. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: SPAA, pp. 34–41. ACM (1995)
27. Peled, D.: All from one, one for all, on model-checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
28. Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: ACSD. IEEE (2012)
29. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multithreaded programs. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 166–182. Springer, Heidelberg (2007)
30. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Comm. of the ACM 53(7), 89–97 (2010)
31. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. on Programming Languages and Systems 10(2), 282–312 (1988)
32. SPARC International, Inc. The SPARC Architecture Manual Version 9 (1994)
33. Tasharofi, S., Karmani, R.K., Lauterburg, S., Legay, A., Marinov, D., Agha, G.: TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 219–234. Springer, Heidelberg (2012)
34. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
35. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: ICSE, pp. 221–230. ACM (2011)
36. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: IPDPS. IEEE (2004)