Acta Informatica (2017) 54:789–818
DOI 10.1007/s00236-016-0275-0

ORIGINAL ARTICLE

Stateless model checking for TSO and PSO

Parosh Aziz Abdulla[1] · **Stavros Aronis**[1] · **Mohamed Faouzi Atig**[1] · **Bengt Jonsson**[1] · **Carl Leonardsson**[1] · **Konstantinos Sagonas**[1]

Received: 7 October 2015 / Accepted: 30 June 2016 / Published online: 7 July 2016

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. This article is an extended version of Abdulla et al. (Tools and

This research was in part funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC). The research time of Stavros Aronis and Konstantinos Sagonas was supported in part by the EU FP7 STREP project RELEASE (287510).

✉ Carl Leonardsson
carl.leonardsson@it.uu.se

Parosh Aziz Abdulla
parosh@it.uu.se

Stavros Aronis
stavros.aronis@it.uu.se

Mohamed Faouzi Atig
mohamed_faouzi.atig@it.uu.se

Bengt Jonsson
bengt.jonsson@it.uu.se

Konstantinos Sagonas
konstantinos.sagonas@it.uu.se

1 Department of Information Technology, Uppsala University, Box 337, 751 05 Uppsala, Sweden

algorithms for the construction and analysis of systems, Springer, New York, pp 353–367, 2015), appearing in TACAS 2015.

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 [4,36], 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) [18], also known as *systematic concurrency testing* [24,39]. 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 [19], Chess [28], 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 *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 [13,17,30,38]. POR has been adapted to SMC in the form of *Dynamic Partial Order Reduction* (DPOR) [16], which has been further developed in recent years [1,22,24,32,33,37]. 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* [27]). DPOR algorithms guarantee to explore at least one execution in each equivalence class, thus attaining full coverage with reduced cost. A recent optimal algorithm [1] 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 [3,5,29]. 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 [35], here called *Shasha–Snir traces*, which is often used in model checking and runtime verification [7,8,10,11,21,23].

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 [1] 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 (as in e.g. [7,8,10,11]), 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 [1], which is simpler than Optimal-DPOR, but about equally effective. Our experimental results for analyses under SC, TSO and PSO of a number of intensely racy benchmarks and programs written in C/pthreads, show 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 [6] and goto-instrument [5], on a number of terminating and data-deterministic benchmarks.

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 $\times 86$ 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 may contain cycles, 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

Fig. 1 A program implementing the classic idiom of Dekker's mutual exclusion algorithm [15]

```
      p                 q
store: x :=1      store: y:=1
load: $r:=y       load: $s:=x
```

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

```
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
```

undesirable distinctions between executions, and how our new representation of chronological traces removes 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.

TSO: an Introduction TSO relaxes the ordering between stores and subsequent loads to different memory locations. This can be modelled operationally by equipping each thread with a *store buffer* [34], 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 most recent such 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, but with the roles of **x** and **y** reversed. All memory locations and registers are assumed to have initial values 0. 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 [35] 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

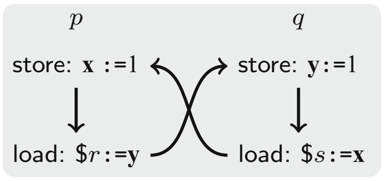


Fig. 3 The Shasha–Snir trace corresponding to the execution in Fig. 2

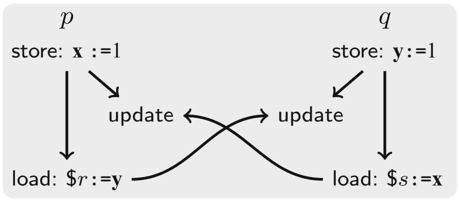


Fig. 4 A trace for the execution in Fig. 2 where updates are separated from stores

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 $i \rightarrow j$ 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.

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 := \mathbf{y}$ and q : store: $\mathbf{y} := 1$, 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.

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

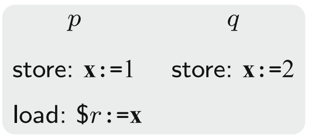


Fig. 5 A program illustrating buffer forwarding

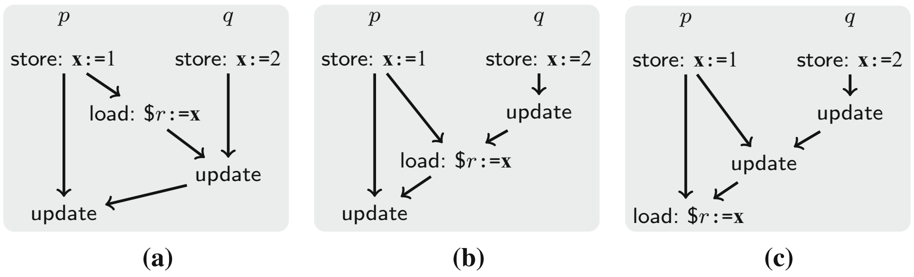


Fig. 6 Three redundant happens-before relations for Fig. 5

Shasha–Snir traces. The reason for this is the mechanism of 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 difficulties for a happens-before relation that orders any update with any load 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. Figure 6a–c shows three of the corresponding happens-before relations. In each of the three cases, the load reads the value 1, written to **x** by p. In Fig. 6a, b the load is satisfied by buffer forwarding, and in Fig. 6c by a read from memory. These three relations all correspond to the same Shasha–Snir trace, shown in Fig. 7a, 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.

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 directly 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 than changing a load from memory into a load of the same value from buffer, as seen when comparing Fig. 6b, c.
2. A load ld from a memory location **x** by a thread p is never directly 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. When we compare Fig. 6a, 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. 6a–c merge into a single representation shown in Fig. 7b. 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

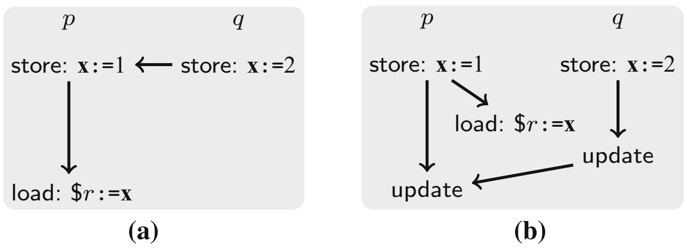


Fig. 7 Traces that capture all three Fig. 6a–c. **a** A Shasha–Snir trace corresponding to all three traces of Fig. 6. **b** The three traces can be merged into this single trace

different values being loaded by the load instruction or different values in memory at the end of the execution. We show in Theorem 1 of Sect. 3 that our proposed representation is in general optimal.

Chronological traces for PSO The TSO and PSO memory models are very similar. The difference is that PSO does not enforce program order between stores by the same thread to different memory locations. To capture this, chronological traces are constructed differently under TSO and PSO. In particular, under TSO there will always be edges between all updates of the same thread, but under PSO we omit those edges when the updates access different memory locations. In Sect. 5 we describe in more detail how to adapt the chronological traces described above to the PSO memory model.

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. This example shows only the intuition of the process, and is intentionally vague. A detailed description of the algorithm is given in Sect. 4.

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. 8a, 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. 8a, the reversible edges are the ones from p : update to q : update, and from p : load: $\$r := \mathbf{x}$ 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: $\$r := \mathbf{x}$ to q : update will generate the execution and chronological trace in Fig. 8b. 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. 8a, 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. 8c.

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

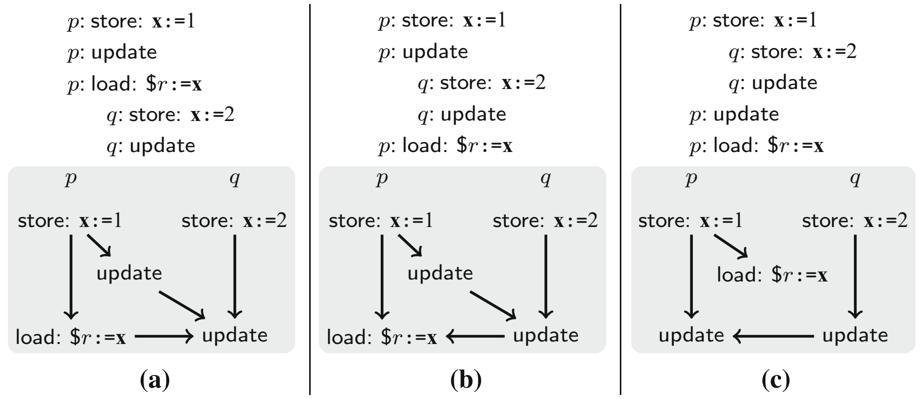


Fig. 8 How SMC with DPOR explores the program of Fig. 5

3 Formalization

In this section we summarize our formalization of the concepts of Sect. 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.

Preliminaries For a function f, we use the notation $f[x \hookleftarrow v]$ to denote the function f' such that $f'(x) = v$ and $f'(y) = f(y)$ whenever $y \neq x$. We use $w \cdot w'$ to denote the concatenation of the words w and w'.

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, $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ 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 the short forms st($\mathbf{x}$) and ld($\mathbf{x}$) to denote some store and load of $\mathbf{x}$ respectively, where the value is not interesting. We use TID to denote the set of all thread identifiers, and MemLoc to denote the set of all memory locations.

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 $\mathbb{L}(p) = (\mathbb{R}, \mathbb{B})$. Here $\mathbb{R}$ is the state of local registers (their valuation denoted $\mathbb{R}(\$r)$) 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 (i.e., most recently inserted) 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 will introduce the notion of *auxiliary threads*. For each thread $p \in$ TID, we assume that there is an auxiliary thread upd(p). The auxiliary thread upd(p) will nondeterministically perform memory updates from the store buffer of p, when the buffer is non-empty. We use

$\mathsf{AuxTID} = \{\mathsf{upd}(p) \big| p \in \mathsf{TID}\}$ to denote the set of auxiliary thread identifiers. We will 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'. We define the transition relation $c \xrightarrow{p} c'$ depending on what the next instruction op of p is in c. In the following we assume $c = (\mathbb{L}, \mathbb{M})$ and $c' = (\mathbb{L}', \mathbb{M}')$ and $\mathbb{L}(p) = (\mathbb{R}, \mathbb{B})$. Let $\mathbb{R}_{\mathsf{pc}}$ 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 $\mathsf{store}\text{:}\ \mathbf{x}{:=}\$r$, then $c \xrightarrow{p} c'$ iff $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}, \mathbb{B} \cdot (\mathbf{x}, v))]$ where $v = \mathbb{R}(\$r)$ and $\mathbb{M}' = \mathbb{M}$ and. Intuitively, under TSO, instead of updating the memory with the new value v, we insert the entry $(\mathbf{x}, v)$ at the end of the store buffer of the thread.

Load If op has the form $\mathsf{load}\text{:}\ \$r{:=}\mathbf{x}$, then $c \xrightarrow{p} c'$ iff $\mathbb{M}' = \mathbb{M}$ and either

1. (**From memory**) $\mathbb{B}(\mathbf{x}) = \bot$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}[\$r \hookleftarrow \mathbb{M}(\mathbf{x})], \mathbb{B})]$, or
2. (**Buffer forwarding**) $\mathbb{B}(\mathbf{x}) \neq \bot$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}[\$r \hookleftarrow \mathbb{B}(\mathbf{x})], \mathbb{B})]$.

Intuitively, in the first case there is no entry for $\mathbf{x}$ in the thread's own store buffer, so the value is read from memory. In the second case, we read the value of $\mathbf{x}$ from its *latest* entry in the store buffer of the thread.

Fence If op has the form fence, then $c \xrightarrow{p} c'$ iff $\mathbb{B} = \varepsilon$ and $\mathbb{L}' = \mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}, \mathbb{B})]$ and $\mathbb{M}' = \mathbb{M}$. 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 say that $c \xrightarrow{\mathsf{upd}(p)} c'$ iff $\mathbb{B} = (\mathbf{x}, v) \cdot \mathbb{B}'$ for some $\mathbf{x}$, 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 Based on the operational semantics defined above, a program execution can be defined as a sequence $c_0 \xrightarrow{p_0} c_1 \xrightarrow{p_1} \cdots \xrightarrow{p_{n-1}} c_n$ 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. Formally, we will therefore define each *execution* as a word of *events*. Each event is a triple (p, i, j) which represents one transition in the run. Here the thread $p \in \mathsf{TID} \cup \mathsf{AuxTID}$ is a regular or auxiliary thread, executing an instruction i (which may be an update $\mathsf{u}(\mathbf{x})$). The natural number j is used to disambiguate events. We let j be such that (p, i, j) is the j:th event of p in the execution (counting from 1). For an event $e = (p, i, j)$, we define $\mathsf{tid}(e) = p$. We will use Event to denote the set of all possible events. Figure 9 shows three sample executions.

For an execution τ and two events e, e' in τ, we say that $e <_\tau e'$ iff e strictly precedes e' in τ. We define two dummy events $e^0 = (\bot, \bot, 0)$ and $e^\infty = (\bot, \bot, \infty)$, and we extend $<_\tau$ such that for every event $e \notin \{e^0, e^\infty\}$ we have $e^0 <_\tau e <_\tau e^\infty$.

For an execution τ and an event $e = (p, \mathsf{st}(\mathbf{x}), j)$ in τ, we define $\mathsf{upd}_{\mathsf{st}}(e)$ to be the update event in τ corresponding to the store event e. Formally, let k be the number of events $e_w = (p', \mathsf{st}(\mathbf{y}), j')$ for any memory location $\mathbf{y}$ in τ such that $p' = p$ and $j' \leq j$. Then

e_s: $(p,\mathsf{st}(\mathbf{x}),1)$
$(p,\mathsf{st}(\mathbf{y}),2)$
$(q,\mathsf{st}(\mathbf{z}),1)$
e_l: $(p,\mathsf{ld}(\mathbf{x}),3)$
e_u: $(\mathsf{upd}(p),\mathsf{u}(\mathbf{x}),1)$
$(\mathsf{upd}(q),\mathsf{u}(\mathbf{z}),1)$
$(\mathsf{upd}(p),\mathsf{u}(\mathbf{y}),2)$

(a) $\mathsf{upd}_{\mathsf{st}}(e_s) = \mathsf{upd}_{\mathsf{ld}}(e_l) = e_u$
Here e_u is the update corresponding to the store e_s. Since e_s is the latest store to **x**, which precedes e_l in program order, we have that $\mathsf{upd}_{\mathsf{ld}}(e_l) = e_u$.

e_s: $(p,\mathsf{st}(\mathbf{x}),1)$
$(q,\mathsf{st}(\mathbf{z}),1)$
e_l: $(p,\mathsf{ld}(\mathbf{x}),2)$
$(\mathsf{upd}(q),\mathsf{u}(\mathbf{z}),1)$

(b) $\mathsf{upd}_{\mathsf{st}}(e_s) = \mathsf{upd}_{\mathsf{ld}}(e_l) = e^{\infty}$
Here the update of e_s is still pending. Therefore we define $\mathsf{upd}_{\mathsf{st}}(e_s)$ as e^{∞}. Since the same pending update originates in a store preceding e_l in program order, we also have $\mathsf{upd}_{\mathsf{ld}}(e_l) = e^{\infty}$.

e_s: $(p,\mathsf{st}(\mathbf{z}),1)$
e_l: $(p,\mathsf{ld}(\mathbf{x}),2)$
e_u: $(\mathsf{upd}(p),\mathsf{u}(\mathbf{z}),1)$

(c) $\mathsf{upd}_{\mathsf{st}}(e_s) = e_u \quad \mathsf{upd}_{\mathsf{ld}}(e_l) = e^0$
Here e_s corresponds to e_u as in Figure 9(a). The load e_l is not preceded by any store to the same memory location. Therefore, we define $\mathsf{upd}_{\mathsf{ld}}(e_l) = e^0$.

Fig. 9 Illustration of the definitions of $\mathsf{upd}_{\mathsf{st}}$ and $\mathsf{upd}_{\mathsf{ld}}$

$\mathsf{upd}_{\mathsf{st}}(e) = (\mathsf{upd}(p), \mathsf{u}(\mathbf{x}), k)$ if there is such an event in τ. Otherwise $\mathsf{upd}_{\mathsf{st}}(e) = e^{\infty}$, denoting that the update is still pending at the end of τ. Figure 9a illustrates the typical case, where the store e_s is eventually followed by its corresponding update $\mathsf{upd}_{\mathsf{st}}(e_s) = e_u$. Figure 9b shows the case when the update corresponding to the store e_s is still pending at the end of the execution, and therefore $\mathsf{upd}_{\mathsf{st}}(e_s) = e^{\infty}$.

For an execution τ and an event $e = (p, \mathsf{ld}(\mathbf{x}), j)$ in τ, we define $\mathsf{upd}_{\mathsf{ld}}(e)$ to be the update event of the latest store to **x**, which precedes e in the same thread. The intuition is that $\mathsf{upd}_{\mathsf{ld}}(e)$ is the update from which e would get its value in the case of buffer forwarding. Formally, if there is an event $e_w = (p, \mathsf{st}(\mathbf{x}), k)$ in τ such that $k < j$ and there is no event $(p, \mathsf{st}(\mathbf{x}), l)$ in τ with $k < l < j$, then $\mathsf{upd}_{\mathsf{ld}}(e) = \mathsf{upd}_{\mathsf{st}}(e_w)$. Otherwise $\mathsf{upd}_{\mathsf{ld}}(e) = e^0$. Figure 9a, b shows the typical case, where $\mathsf{upd}_{\mathsf{ld}}(e_l)$ is taken to be the update corresponding to the latest preceding store by the same thread to the same memory location. Figure 9c shows the case when there is no such preceding store, and $\mathsf{upd}_{\mathsf{ld}}(e_l)$ is taken to be the dummy event e^0. (Notice that the store e_s is to a different memory location.)

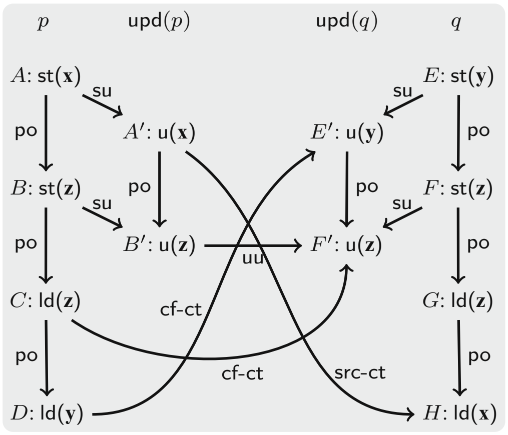
Chronological traces We can now introduce the main conceptual contribution of the paper, viz. *chronological traces*. For an execution τ we define its chronological trace $\mathcal{T}_C(\tau)$ as a directed graph $\langle V, E\rangle$. The vertices V are all the events in τ; both events representing instructions and events representing updates. The set of edges E is the union of six relations: $E = \rightarrow_{\tau}^{\mathsf{po}} \cup \rightarrow_{\tau}^{\mathsf{su}} \cup \rightarrow_{\tau}^{\mathsf{uu}} \cup \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} \cup \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} \cup \rightarrow_{\tau}^{\mathsf{uf}}$.

We will illustrate the definition on an execution of the program in Fig. 10a, which contains an idiom that occurs in the mutual exclusion algorithm of Peterson [31]. It is mostly the same as that from Dekker's mutual exclusion algorithm. But it has two additional accesses in each thread to a separate memory location **z**. These provide an opportunity to display buffer forwarding. Figure 10c shows an example of an execution and Fig. 10b shows its corresponding chronological trace.

We define the edge relations of chronological traces as follows, for two arbitrary events $e = (p, i, j) \in V$ and $e' = (p', i', j') \in V$:

p	q
A: store: $\mathbf{x}$:=1	E: store: $\mathbf{y}$:=1
B: store: $\mathbf{z}$:=1	F: store: $\mathbf{z}$:=0
C: load: \$$r$:=$\mathbf{z}$	G: load: \$$r$:=$\mathbf{z}$
D: load: \$$s$:=$\mathbf{y}$	H: load: \$$s$:=$\mathbf{x}$

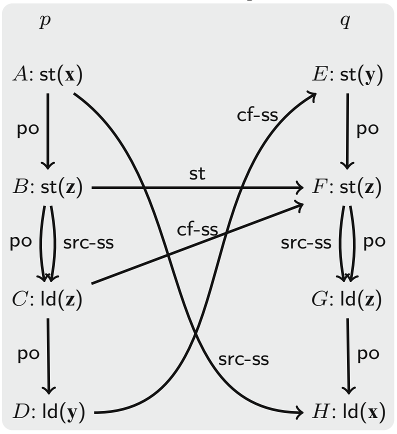
(a) A small program illustrating the idiom of Peterson's mutual exclusion algorithm.



(b) The chronological trace $\mathcal{T}_C(\tau)$ corresponding to the execution in Figure 10(c). Notice that there is no edge between the load G from **z** and either of the updates to **z**.

A: (p, st($\mathbf{x}$),1)
    E: (q, st($\mathbf{y}$),1)
B: (p, st($\mathbf{z}$),2)
C: (p, ld($\mathbf{z}$),3)
D: (p, ld($\mathbf{y}$),4)
    E': (upd(q), u($\mathbf{y}$),1)
    F: (q, st($\mathbf{z}$),2)
    G: (q, ld($\mathbf{z}$),3)
A': (upd(p), u($\mathbf{x}$),1)
    H: (q, ld($\mathbf{x}$),4)
B': (upd(p), u($\mathbf{z}$),2)
    F': (upd(q), u($\mathbf{z}$),2)

(c) An execution τ.



(d) The Shasha-Snir trace $\mathcal{T}(\tau)$ corresponding to the execution in Figure 10(c).

Fig. 10 Traces illustrated by the idiom of Peterson's mutual exclusion algorithm

Program order $e \rightarrow_{\tau}^{\mathsf{po}} e'$ iff $p = p'$ and $j' = j + 1$. For example, we see in Fig. 10b that there is a program order edge from the store instruction A (i.e., the event $(p, \mathsf{st}(x), 1)$) to the store instruction B (i.e., the event $(p, \mathsf{st}(z), 2)$) which immediately follows it in the program of thread p. Similarly, the updates of each thread are program ordered. E.g., $A' \rightarrow_{\tau}^{\mathsf{po}} B'$.

Store to update $e \rightarrow_{\tau}^{\mathsf{su}} e'$ iff $i = \mathsf{st}(\mathbf{x})$ for some $\mathbf{x}$ and $\mathsf{upd}_{\mathsf{st}}(e) = e'$. I.e., e' is the update corresponding to the store e. This is illustrated in Fig. 10b where there is an su-edge from each store, to its corresponding update. E.g., $A \rightarrow_{\tau}^{\mathsf{su}} A'$.

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 $e <_{\tau} e'$ and there is no event $e'' = (p'', \mathsf{u}(\mathbf{x}), j'')$ such that $e <_{\tau} e'' <_{\tau} e'$. I.e., $\rightarrow_{\tau}^{\mathsf{uu}}$ chronologically orders all updates for each memory location. In Fig. 10b we see that the two updates B' and F' to **z** are uu-ordered with each other in the same order as they appear in the execution in Fig. 10c. However, they are not uu-ordered with the updates A' and E' to $\mathbf{x}$ and $\mathbf{y}$.

Source $e \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} e'$ iff for some $\mathbf{x}$ it holds that $i = \mathsf{u}(\mathbf{x})$ and $i' = \mathsf{ld}(\mathbf{x})$ and $\mathsf{upd}_{\mathsf{ld}}(e') <_{\tau} e <_{\tau} e'$ and there is no update $e'' = (p'', \mathsf{u}(\mathbf{x}), j'')$ to $\mathbf{x}$ such that $e <_{\tau} e'' <_{\tau} e'$. I.e., if the source of the value read by e' is an update e from a different process, then $e \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} e'$. Otherwise, there is no incoming $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}}$ edge to e'. Since the definition forces the strict order $\mathsf{upd}_{\mathsf{ld}}(e') <_{\tau} e <_{\tau} e'$, it excludes the possibility of the update e originating in the same thread as the load e' (as no update from p' can come after $\mathsf{upd}_{\mathsf{ld}}(e')$ but before e'). Therefore a load is never $\mathsf{src\text{-}ct}$-related to an update from the same thread. In Fig. 10b we see that the load H takes its value from the update A'. Therefore the events are $\mathsf{src\text{-}ct}$-related. But the loads C and G to $\mathbf{z}$ both read the value written by their own thread, and therefore have no $\mathsf{src\text{-}ct}$-relation. The "ct" in the name of the relation stands for "chronological trace", and serves to distinguish the relation $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}}$ for chronological traces from the similar, but different relation $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}}$ for Shasha–Snir traces (introduced below).

Conflict $e \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} e'$ iff $i = \mathsf{ld}(\mathbf{x})$ and $i' = \mathsf{u}(\mathbf{x})$ for some $\mathbf{x}$ and e' is the first (w.r.t. $<_{\tau}$) event e_u of the form $(\_, \mathsf{u}(\mathbf{x}), \_)$ such that both $e <_{\tau} e_u$ and $\mathsf{upd}_{\mathsf{ld}}(e) <_{\tau} e_u$. The intuition here is that $e \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} e'$ when e' is the first update which succeeds e in the coherence order of $\mathbf{x}$. Equivalently, e' is the update that overwrites the value that was read by e. In Fig. 10b, the load D to $\mathbf{y}$ by p reads the initial value of $\mathbf{y}$, which is then overwritten by the update E' to $\mathbf{y}$ by q. Therefore we have $D \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} E'$. The load C reads the value of the update B' by buffer forwarding. That value is later overwritten in memory by the update F'. Therefore we have $C \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} F'$.

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 $e <_{\tau} e'$ and there is no event $e'' = (p, \mathsf{u}(\mathbf{y}), j'')$ for any $\mathbf{y}$ such that $e <_{\tau} e'' <_{\tau} e'$. 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.

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. We say that an execution τ is completed when all stores have reached memory, i.e., when for every event $e = (p, \mathsf{st}(\mathbf{x}), j)$ in τ we have $\mathsf{upd}_{\mathsf{st}}(e) \neq e^{\infty}$. 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 where $i \neq \mathsf{u}(\mathbf{x})$ for all $\mathbf{x}$. The edges E is the union of four relations $E = \rightarrow_{\tau}^{\mathsf{po}} \cup \rightarrow_{\tau}^{\mathsf{st}} \cup \rightarrow_{\tau}^{\mathsf{src\text{-}ss}} \cup \rightarrow_{\tau}^{\mathsf{cf\text{-}ss}}$.

For two arbitrary events $e = (p, i, j) \in V$ and $e' = (p', i', j') \in V$, we define the relations as follows:

Program order $e \rightarrow_{\tau}^{\mathsf{po}} e'$ iff $p = p'$ and $j' = j + 1$. This is the same program order as for chronological traces.

Store order $e \rightarrow_{\tau}^{\mathsf{st}} e'$ iff $i = \mathsf{st}(\mathbf{x})$ and $i' = \mathsf{st}(\mathbf{x})$ for some $\mathbf{x}$ and the corresponding updates are ordered in τ s.t. $\mathsf{upd}_{\mathsf{st}}(e) <_{\tau} \mathsf{upd}_{\mathsf{st}}(e')$ and there is no other update event $e'' = (p'', \mathsf{u}(\mathbf{x}), j'')$

such that $\mathsf{upd}_{\mathsf{st}}(e) <_\tau e'' <_\tau \mathsf{upd}_{\mathsf{st}}(e')$. I.e., for each memory location $\mathbf{x}$, the transitive closure $\rightarrow_\tau^{\mathsf{st}*}$ is a total order on all stores to $\mathbf{x}$ based on the order in which they reach memory.

Source $e \rightarrow_\tau^{\mathsf{src\text{-}ss}} e'$ iff $i' = \mathsf{ld}(\mathbf{x})$ and e is the maximal store event $e'' = (p'', \mathsf{st}(\mathbf{x}), j'')$ with respect to $\rightarrow_\tau^{\mathsf{st}*}$ such that either $\mathsf{upd}_{\mathsf{st}}(e'') <_\tau e'$ or $e'' \rightarrow_\tau^{\mathsf{po}*} e'$. I.e., $e \rightarrow_\tau^{\mathsf{src\text{-}ss}} e'$ when 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 $i = \mathsf{ld}(\mathbf{x})$ and $i' = \mathsf{st}(\mathbf{x})$ and if there is an event e'' such that $e'' \rightarrow_\tau^{\mathsf{src\text{-}ss}} e$ then $e'' \rightarrow_\tau^{\mathsf{st}} e'$, otherwise e' has no predecessor in $\rightarrow_\tau^{\mathsf{st}}$. I.e., e' is the store which overwrites the value that was read by e.

The definition of Shasha–Snir traces is illustrated in Fig. 10d. 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 $\mathcal{T}_C(\tau) = \mathcal{T}_C(\tau')$.*

We decompose the theorem into the following two lemmas, which are proven separately.

Lemma 1 (Equivalence of Shasha–Snir traces and chronological traces $\Rightarrow$ direction) *For a given program $\mathcal{P}$ with two completed executions τ, τ', it holds that if $\mathcal{T}(\tau) = \mathcal{T}(\tau')$ then $\mathcal{T}_C(\tau) = \mathcal{T}_C(\tau')$.*

Lemma 2 (Equivalence of Shasha–Snir traces and chronological traces $\Leftarrow$ direction) *For a given program $\mathcal{P}$ with two completed executions τ, τ', it holds that if $\mathcal{T}_C(\tau) = \mathcal{T}_C(\tau')$ then $\mathcal{T}(\tau) = \mathcal{T}(\tau')$.*

Proof of Lemma 1 Let two completed executions τ and τ' be given. Let

$$\begin{aligned}
\mathcal{T}(\tau) &= \langle V_{SS}, \rightarrow_\tau^{\mathsf{po}} \cup \rightarrow_\tau^{\mathsf{st}} \cup \rightarrow_\tau^{\mathsf{src\text{-}ss}} \cup \rightarrow_\tau^{\mathsf{cf\text{-}ss}} \rangle \text{ and} \\
\mathcal{T}(\tau') &= \langle V'_{SS}, \rightarrow_{\tau'}^{\mathsf{po}} \cup \rightarrow_{\tau'}^{\mathsf{st}} \cup \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}} \cup \rightarrow_{\tau'}^{\mathsf{cf\text{-}ss}} \rangle \text{ and } \mathcal{T}_C(\tau) \\
&= \langle V_C, \rightarrow_\tau^{\mathsf{po}} \cup \rightarrow_\tau^{\mathsf{su}} \cup \rightarrow_\tau^{\mathsf{uu}} \cup \rightarrow_\tau^{\mathsf{src\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{cf\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{uf}} \rangle \text{ and} \\
\mathcal{T}_C(\tau') &= \langle V'_C, \rightarrow_{\tau'}^{\mathsf{po}} \cup \rightarrow_{\tau'}^{\mathsf{su}} \cup \rightarrow_{\tau'}^{\mathsf{uu}} \cup \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}} \cup \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}} \cup \rightarrow_{\tau'}^{\mathsf{uf}} \rangle.
\end{aligned}$$

Furthermore, assume that $\mathcal{T}(\tau) = \mathcal{T}(\tau')$.

First, we determine that the events are the same in both chronological traces: $V_C = V'_C$. From $V_{SS} = V'_{SS}$ we have that the non-update events in τ are the same as the ones in τ'. Since τ and τ' contain the same stores for each thread in the same per-thread order, it follows from the completedness of τ and τ', and from the TSO semantics that τ and τ' also have the same update events. Hence $V_C = V'_C$.

We see that the definitions of program order and store to update order in chronological traces are entirely determined by which events exist in the execution for each thread. Since both executions have the same events, we conclude that $\rightarrow_\tau^{\mathsf{po}} = \rightarrow_{\tau'}^{\mathsf{po}}$ and $\rightarrow_\tau^{\mathsf{su}} = \rightarrow_{\tau'}^{\mathsf{su}}$. The equality $\rightarrow_\tau^{\mathsf{uf}} = \rightarrow_{\tau'}^{\mathsf{uf}}$ of update to fence order follows similarly.

Let us consider the definitions of update to update order for chronological traces and store order for Shasha–Snir traces. We see that there is a one-to-one mapping between relations $e \rightarrow_\tau^{\mathsf{st}} e'$ for stores in Shasha–Snir traces to relations $\mathsf{upd}_{\mathsf{st}}(e) \rightarrow_\tau^{\mathsf{uu}} \mathsf{upd}_{\mathsf{st}}(e')$ in chronological traces. Since the store orders are the same for τ and τ', we thus conclude that the update to update orders are also the same: $\rightarrow_\tau^{\mathsf{uu}} = \rightarrow_{\tau'}^{\mathsf{uu}}$.

We now turn our attention to proving that $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$. We will first prove that $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}} \subseteq \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$. From symmetry it then follows that $\rightarrow_{\tau'}^{\mathsf{src\text{-}ct}} \subseteq \rightarrow_{\tau}^{\mathsf{src\text{-}ct}}$, and hence that $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$. Let us assume that the relation $e \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} e'$ exists in $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}}$ for some events $e = (p, \mathsf{u}(\mathbf{x}), j)$ and $e' = (p', \mathsf{ld}(\mathbf{x}), j')$. We will prove that the same relation $e \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}} e'$ exists in $\rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$. From the definition of $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}}$ we have that $\mathsf{upd}_{\mathsf{ld}}(e') <_{\tau} e <_{\tau} e'$ and there is no update $e'' = p'', \mathsf{u}(\mathbf{x}), i'')$ to the same memory location such that $e <_{\tau} e'' <_{\tau} e'$. Since e' is preceded in τ by at least one update to $\mathbf{x}$, there must be a store event e_w such that $e_w \rightarrow_{\tau}^{\mathsf{src\text{-}ss}} e'$ in τ. From the definition of $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}}$ we have that e_w is the maximal event $(p'', \mathsf{st}(\mathbf{x}), j'')$ with respect to $\rightarrow_{\tau}^{\mathsf{st}*}$ such that either $\mathsf{upd}_{\mathsf{st}}(e_w) <_{\tau} e'$ or $e_w \rightarrow_{\tau}^{\mathsf{po}*} e'$. If $e_w \rightarrow_{\tau}^{\mathsf{po}*} e'$, then $\mathsf{upd}_{\mathsf{st}}(e_w) = \mathsf{upd}_{\mathsf{ld}}(e')$. But then the maximality of e_w contradicts $\mathsf{upd}_{\mathsf{ld}}(e') <_{\tau} e <_{\tau} e'$. Hence we have $\mathsf{upd}_{\mathsf{st}}(e_w) <_{\tau} e'$. Maximality of e_w now gives that $\mathsf{upd}_{\mathsf{st}}(e_w) = e$. Since $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$ we have that in τ' also $e_w \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}} e'$. From the definition of $\rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$ and $\neg(e_w \rightarrow_{\tau'}^{\mathsf{po}*} e')$ we know that $\mathsf{upd}_{\mathsf{st}}(e_w)$ is the store-order-maximal update to $\mathbf{x}$ that precedes e' in τ'. Since the store order is the same for τ and τ' we have $\mathsf{upd}_{\mathsf{ld}}(e') <_{\tau'} e$. But then $e = \mathsf{upd}_{\mathsf{st}}(e_w)$ satisfies the criteria for $e \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}} e'$.

Finally, we will show that $\rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} = \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}}$. Similarly to the proof for $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$, it suffices here to show that $\rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} \subseteq \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}}$. Assume therefore that $e_r \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} e_u$ for some events $e_r = (p, \mathsf{ld}(\mathbf{x}), j)$, $e_u = (p', \mathsf{u}(\mathbf{x}), j')$. We will show that $e_r \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}} e_u$. The definition of $\rightarrow_{\tau}^{\mathsf{cf\text{-}ct}}$ gives that e_u is the first (w.r.t. $<_{\tau}$) event e of the form $(\_, \mathsf{u}(\mathbf{x}), \_)$ such that both $e_r <_{\tau} e$ and $\mathsf{upd}_{\mathsf{ld}}(e_r) <_{\tau} e$. Let e_w be the store event such that $\mathsf{upd}_{\mathsf{st}}(e_w) = e_u$. We will split the proof in cases depending on whether or not there exists a source event for e_r in the Shasha–Snir traces.

Assume therefore first (i) that there is no event e_{src} such that $e_{src} \rightarrow_{\tau}^{\mathsf{src\text{-}ss}} e_r$. Then there is no update to $\mathbf{x}$ that precedes e_r in $<_{\tau}$. Furthermore $\mathsf{upd}_{\mathsf{ld}}(e_r) = e^0$. This tells us that e_w has no predecessor in $\rightarrow_{\tau}^{\mathsf{st}}$. Since $\rightarrow_{\tau}^{\mathsf{st}} = \rightarrow_{\tau'}^{\mathsf{st}}$, we also have that e_w has no predecessor in $\rightarrow_{\tau'}^{\mathsf{st}}$. Furthermore, since e_r has no source event in τ', it must be the case that $e_r <_{\tau'} e_u$. But then, e_u is the first update event in τ' which is after both e_r and $\mathsf{upd}_{\mathsf{ld}}(e_r)$. And so we have $e_r \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}} e_u$.

Next assume (ii) that there is an event e_{src} with $e_{src} \rightarrow_{\tau}^{\mathsf{src\text{-}ss}} e_r$ and that $\mathsf{tid}(e_{src}) = \mathsf{tid}(e_r)$. Then it must be the case that $\mathsf{upd}_{\mathsf{st}}(e_{src}) = \mathsf{upd}_{\mathsf{ld}}(e_r)$. Since $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$, we have that $e_{src} \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}} e_r$. There can be no update event e to the same memory location $\mathbf{x}$ such that $\mathsf{upd}_{\mathsf{ld}}(e_r) <_{\tau} e <_{\tau} e_r$. If there were such an e, then e_{src} wouldn't be the source of e_r. The same argument goes in τ'. This tells us that e_u is the immediate store order successor of $\mathsf{upd}_{\mathsf{ld}}(e_r)$, i.e., $\mathsf{upd}_{\mathsf{ld}}(e_r) \rightarrow_{\tau}^{\mathsf{uu}} e_u$ and $e_{src} \rightarrow_{\tau}^{\mathsf{st}} e_w$. Since $\rightarrow_{\tau}^{\mathsf{uu}} = \rightarrow_{\tau'}^{\mathsf{uu}}$, we have $\mathsf{upd}_{\mathsf{ld}}(e_r) \rightarrow_{\tau'}^{\mathsf{uu}} e_u$. Hence e_u is the first update event which succeeds both e_r and $\mathsf{upd}_{\mathsf{ld}}(e_r)$ in $<_{\tau'}$. Thus $e_r \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}} e_u$.

Lastly, we assume (iii) that there is an event e_{src} such that $e_{src} \rightarrow_{\tau}^{\mathsf{src\text{-}ss}} e_r$ and that $\mathsf{tid}(e_{src}) \neq \mathsf{tid}(e_r)$. Then it is the case in τ that $\mathsf{upd}_{\mathsf{ld}}(e_r) <_{\tau} \mathsf{upd}_{\mathsf{st}}(e_{src}) <_{\tau} e_r$. And there is no update event e to $\mathbf{x}$ such that $\mathsf{upd}_{\mathsf{st}}(e_{src}) <_{\tau} e <_{\tau} e_r$. The same holds in τ'. Since e_u is the first update to $\mathbf{x}$ after e_r in τ, this means that we have $\mathsf{upd}_{\mathsf{st}}(e_{src}) \rightarrow_{\tau}^{\mathsf{uu}} e_u$. We have $\rightarrow_{\tau}^{\mathsf{uu}} = \rightarrow_{\tau'}^{\mathsf{uu}}$, so $\mathsf{upd}_{\mathsf{st}}(e_{src}) \rightarrow_{\tau'}^{\mathsf{uu}} e_u$. Now it must be the case that $e_r <_{\tau'} e_u$. Otherwise, e_{src} wouldn't be the source of e_r in τ', and we know $e_{src} \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}} e_r$. Hence e_u is an update event that succeeds both e_r and $\mathsf{upd}_{\mathsf{ld}}(e_r)$ in $<_{\tau'}$. It remains to show that it is the first such update. Suppose $e \neq e_u$ is an update event to $\mathbf{x}$ such that $e_r <_{\tau} e <_{\tau} e_u$. Then it would be the case that $\mathsf{upd}_{\mathsf{st}}(e_{src}) <_{\tau'} e <_{\tau'} e_u$. But this would contradict $\mathsf{upd}_{\mathsf{st}}(e_{src}) \rightarrow_{\tau'}^{\mathsf{uu}} e_u$. Thus we have $e_r \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}} e_u$.

This concludes the proof of $\mathcal{T}_C(\tau) = \mathcal{T}_C(\tau')$. □

Proof of Lemma 2 Let two completed executions τ and τ' be given. Let

$$\mathcal{T}(\tau) = \langle V_{SS}, \rightarrow_{\tau}^{\mathsf{po}} \cup \rightarrow_{\tau}^{\mathsf{st}} \cup \rightarrow_{\tau}^{\mathsf{src\text{-}ss}} \cup \rightarrow_{\tau}^{\mathsf{cf\text{-}ss}} \rangle \text{ and}$$
$$\mathcal{T}(\tau') = \langle V'_{SS}, \rightarrow_{\tau'}^{\mathsf{po}} \cup \rightarrow_{\tau'}^{\mathsf{st}} \cup \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}} \cup \rightarrow_{\tau'}^{\mathsf{cf\text{-}ss}} \rangle \text{ and}$$
$$\mathcal{T}_C(\tau) = \langle V_C, \rightarrow_{\tau}^{\mathsf{po}} \cup \rightarrow_{\tau}^{\mathsf{su}} \cup \rightarrow_{\tau}^{\mathsf{uu}} \cup \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} \cup \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}} \cup \rightarrow_{\tau}^{\mathsf{uf}} \rangle \text{ and}$$
$$\mathcal{T}_C(\tau') = \langle V'_C, \rightarrow_{\tau'}^{\mathsf{po}} \cup \rightarrow_{\tau'}^{\mathsf{su}} \cup \rightarrow_{\tau'}^{\mathsf{uu}} \cup \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}} \cup \rightarrow_{\tau'}^{\mathsf{cf\text{-}ct}} \cup \rightarrow_{\tau'}^{\mathsf{uf}} \rangle.$$

Furthermore, assume that $\mathcal{T}_C(\tau) = \mathcal{T}_C(\tau')$.

We will prove that $\mathcal{T}(\tau) = \mathcal{T}(\tau')$. We know that V_{SS} (respectively V'_{SS}) is precisely the non-updates of V_C (respectively V'_C). Since $V_C = V'_C$ we have $V_{SS} = V'_{SS}$.

For the relations $\rightarrow_{\tau}^{\mathsf{po}}$ and $\rightarrow_{\tau}^{\mathsf{st}}$, a reasoning analogue to that in the $\Rightarrow$ direction gives that $\rightarrow_{\tau}^{\mathsf{po}} = \rightarrow_{\tau'}^{\mathsf{po}}$ and $\rightarrow_{\tau}^{\mathsf{st}} = \rightarrow_{\tau'}^{\mathsf{st}}$.

We will show that $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}} \subseteq \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$. Symmetry then gives $\rightarrow_{\tau'}^{\mathsf{src\text{-}ss}} \subseteq \rightarrow_{\tau}^{\mathsf{src\text{-}ss}}$, and hence $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$. Assume therefore that $e_w \rightarrow_{\tau}^{\mathsf{src\text{-}ss}} e_r$ holds for some events $e_w = (p, \mathsf{st}(\mathbf{x}), j)$ and $e_r = (p', \mathsf{ld}(\mathbf{x}), j')$. Then by the definition of $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}}$ we have that e_w is the maximal event $e = (p'', \mathsf{st}(\mathbf{x}), j'')$ with respect to $\rightarrow_{\tau}^{\mathsf{st}*}$ such that either $\mathsf{upd}_{\mathsf{st}}(e) <_{\tau} e_r$ or $e \rightarrow_{\tau}^{\mathsf{po}*} e_r$. We will separate the proof by cases: either $\mathsf{tid}(e_w) = \mathsf{tid}(e_r)$ or $\mathsf{tid}(e_w) \neq \mathsf{tid}(e_r)$.

Assume first (i) that $\mathsf{tid}(e_w) = \mathsf{tid}(e_r)$. Then it holds that $e_w \rightarrow_{\tau}^{\mathsf{po}*} e_r$, since the events must be program ordered, and the other direction implies $e_r <_{\tau} \mathsf{upd}_{\mathsf{st}}(e_w)$. Program order is the same in τ' as in τ, so we also have $e_w \rightarrow_{\tau'}^{\mathsf{po}*} e_r$. It remains to show that e_w is maximal in τ'. First we conclude that there can be no store event e such that $e_w \rightarrow_{\tau'}^{\mathsf{st}} e$ and $e \rightarrow_{\tau'}^{\mathsf{po}*} e_r$. This is because both the program order and the store order are the same in τ' as in τ, and hence such an event e would contradict the assumed maximality of e_w w.r.t. τ. As a corollary we have $\mathsf{upd}_{\mathsf{ld}}(e_r) = \mathsf{upd}_{\mathsf{st}}(e_w)$. Next we need to conclude that there is no event e such that $e_w \rightarrow_{\tau'}^{\mathsf{st}} e$ and $\mathsf{upd}_{\mathsf{st}}(e) <_{\tau'} e_r$. We know that there is no such event in τ: i.e., there is no event e such that $e_w \rightarrow_{\tau}^{\mathsf{st}} e$ and $\mathsf{upd}_{\mathsf{st}}(e) <_{\tau} e_r$. Hence by the definition of $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}}$ there is no event e_{src}^{C} which is source related with e_r in the chronological trace: $e_{src}^{C} \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} e_r$. Since $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$, the same holds in τ'. Now if there were an event such as e in τ', then e_r would have a source according to $\rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$. This is a contradiction, and so there can be no such e in τ'. Hence, e_w is the maximal store event w.r.t. $\rightarrow_{\tau'}^{\mathsf{st}*}$ which is either updated $<_{\tau'}$-before e_r or program order-before e_r. That concludes the proof for the case that $\mathsf{tid}(e_w) = \mathsf{tid}(e_r)$.

Next assume (ii) that $\mathsf{tid}(e_w) \neq \mathsf{tid}(e_r)$. Clearly e_w is not program ordered with e_r. Hence the definition of $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}}$ gives that $\mathsf{upd}_{\mathsf{st}}(e_w) <_{\tau} e_r$. The maximality of e_w gives that $\mathsf{upd}_{\mathsf{ld}}(e_r) <_{\tau} \mathsf{upd}_{\mathsf{st}}(e_w)$, and that there is no update event $e = (p'', \mathsf{u}(\mathbf{x}), j'')$ such that $\mathsf{upd}_{\mathsf{st}}(e_w) <_{\tau} e <_{\tau} e_r$. Then by the definition of $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}}$ we have $\mathsf{upd}_{\mathsf{st}}(e_w) \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} e_r$. By $\rightarrow_{\tau}^{\mathsf{src\text{-}ct}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$ we also have $\mathsf{upd}_{\mathsf{st}}(e_w) \rightarrow_{\tau'}^{\mathsf{src\text{-}ct}} e_r$. By the definition of $\rightarrow_{\tau'}^{\mathsf{src\text{-}ct}}$ we now have that e_w is the greatest (w.r.t. $<_{\tau'}$) store event with $\mathsf{upd}_{\mathsf{st}}(e_w) <_{\tau'} e_r$. We also have that $\mathsf{upd}_{\mathsf{ld}}(e_r) <_{\tau'} \mathsf{upd}_{\mathsf{st}}(e_w)$. Since there can be no event $e = (\_, \mathsf{st}(\mathbf{x}), \_)$ such that $e \rightarrow_{\tau'}^{\mathsf{po}*} e_r$ and $\mathsf{upd}_{\mathsf{ld}}(e_r) <_{\tau'} \mathsf{upd}_{\mathsf{st}}(e)$, we have that e_w is the maximal event $e = (\_, \mathsf{st}(\mathbf{x}), \_)$ with respect to $\rightarrow_{\tau}^{\mathsf{st}*}$ such that either $\mathsf{upd}_{\mathsf{st}}(e) <_{\tau'} e_r$ or $e \rightarrow_{\tau'}^{\mathsf{po}*} e_r$. Hence $e_w \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}} e_r$. This concludes the proof for $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$.

Since $\rightarrow_{\tau}^{\mathsf{cf\text{-}ss}}$ (respectively $\rightarrow_{\tau'}^{\mathsf{cf\text{-}ss}}$) is entirely determined by $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}}$ and $\rightarrow_{\tau}^{\mathsf{st}}$ (respectively $\rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$ and $\rightarrow_{\tau'}^{\mathsf{st}}$), and we know that $\rightarrow_{\tau}^{\mathsf{src\text{-}ss}} = \rightarrow_{\tau'}^{\mathsf{src\text{-}ss}}$ and $\rightarrow_{\tau}^{\mathsf{st}} = \rightarrow_{\tau'}^{\mathsf{st}}$, we immediately get that $\rightarrow_{\tau}^{\mathsf{cf\text{-}ss}} = \rightarrow_{\tau'}^{\mathsf{cf\text{-}ss}}$. This concludes the proof. □

Proof of Theorem 1 The theorem follows directly from Lemmas 1 and 2. □

4 DPOR algorithm 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 Sect. 2. The explored executions follow the semantics of TSO in Sect. 3. For each execution, its happens-before relation is computed, which is the transitive closure of the edge relation $\rightarrow_\tau^{\mathsf{ct}} = \rightarrow_\tau^{\mathsf{po}} \cup \rightarrow_\tau^{\mathsf{su}} \cup \rightarrow_\tau^{\mathsf{uu}} \cup \rightarrow_\tau^{\mathsf{src\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{cf\text{-}ct}} \cup \rightarrow_\tau^{\mathsf{uf}}$ of the corresponding chronological trace. 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. In this section, we will show concretely how to compute the happens-before relation, and how to use it to instantiate a DPOR algorithm. To do so, we will first need to introduce some further concepts.

The happens-before relation $\rightarrow_\tau^{\mathsf{ct}}$ is computed on the fly, using vector clocks, while taking the particular structure of chronological traces into account. The main difference from computing happens-before relations for sequentially consistent executions (see, e.g., [32]) is that load events which get their value by store forwarding are not immediately synchronized with the vector clock of the memory location. Instead, the load is associated with the store buffer entry from which it got its value. The load is then synchronized with the memory location at the time when the store buffer entry is updated to memory.

Formally, we extend the TSO configurations described in Sect. 3 to keep track of the necessary information about relations between different events. Below we need vector clocks. A *vector clock* is a function $C : (\mathsf{TID} \cup \mathsf{AuxTID}) \mapsto \mathbb{N}$. The intuition is that C captures a set of observed events. For every thread p, the first $C(p)$ events by p have been observed. We let $\mathsf{VecClocks} = ((\mathsf{TID} \cup \mathsf{AuxTID}) \mapsto \mathbb{N})$ denote the set of vector clocks.

For two vector clocks v, v' we use the notation $v \sqcup v'$ to denote the vector clock v'' such that $v''(p) = max(v(p), v'(p))$ for all p. For two vector clocks v, v' we say that $v \leq v'$ when $v(p) \leq v'(p)$ for all p. We say that $v < v'$ if at least one of the inequalities is strict. For an event e and a set E of events we define $E \oplus e = \{e' \in E | \mathsf{tid}(e') \neq \mathsf{tid}(e)\} \cup \{e\}$, i.e. $E \oplus e$ is E where e replaces the previous event $e' \in E$ s.t. $\mathsf{tid}(e') = \mathsf{tid}(e)$. We use the shorthand $f[x_0, x_1, \cdots, x_n \hookleftarrow v]$ to denote $f[x_0 \hookleftarrow v][x_1 \hookleftarrow v] \cdots [x_n \hookleftarrow v]$, i.e., an assignment of the same value to multiple function arguments.

An *extended configuration* is a quintuple $(\mathbb{L}, \mathbb{M}, \mathcal{C}, \mathcal{B}, \mathcal{M})$, where $(\mathbb{L}, \mathbb{M})$ is a TSO configuration as described in Sect. 3.

$\mathcal{C} : (\mathsf{TID} \cup \mathsf{AuxTID} \cup \mathsf{Event} \cup \{\bot\}) \mapsto \mathsf{VecClocks}$
maps each (real or auxiliary) thread identifier p to a vector clock representing which parts of the execution have been seen by p. Also, $\mathcal{C}$ maps each event e to the value of $\mathcal{C}(\mathsf{tid}(e))$ at the time immediately after executing e. We fix that $\mathcal{C}(\bot) = (\lambda x.0)$ is a zeroed clock.
$\mathcal{B} : \mathsf{TID} \mapsto (\mathsf{MemLoc} \times \mathsf{Event} \times (\mathsf{Event} \cup \{\bot\}))^*$
maps each real (not auxiliary) thread ID p to a word of letters $(\mathbf{x}, e_s, e_l)$, each of which keeps an auxiliary state for the corresponding letter in the store buffer in $\mathbb{L}(p)$. Here $\mathbf{x}$ is the accessed memory location, e_s is the store event that produced that letter, and e_l is the latest buffer forwarded load event for which the letter has been the source (if there is no such event then $e_l = \bot$).
$\mathcal{M} : \mathsf{MemLoc} \mapsto ((\mathsf{Event} \cup \{\bot\}) \times 2^{\mathsf{Event}})$
maps each memory location $\mathbf{x}$ to a pair (e_u, E_l), where e_u is the latest update event that accessed $\mathbf{x}$ (or $\bot$ if $\mathbf{x}$ has never been updated), and where E_l is a set which for each thread p that has read $\mathbf{x}$ contains the latest event of p that read the value of $\mathbf{x}$.

Initially all clocks in $\mathcal{C}$ are zeroed, all buffers in $\mathcal{B}$ are empty, and for all memory locations $\mathbf{x}$ we have $\mathcal{M}(\mathbf{x}) = (\bot, \emptyset)$.

The idea here is that as we execute memory accesses, we update the vector clock of the executing thread to reflect which new events have been observed.

For example, when we execute an update $e_{\mathbf{x}}$ corresponding to a buffer entry $(\mathbf{x}, e_s, e_l)$, we look to the memory $\mathcal{M}(\mathbf{x}) = (e_u, E_l)$. We know that the update event is ordered after the previous update e_u, as well as the previous loads in E_l and the store event e_s which enabled the update e_x. We update the vector clock $\mathcal{C}(\mathsf{tid}(e_{\mathbf{x}}))$ of the auxiliary thread to include all these newly observed events.

The procedure for a load from memory is similar, except that we do not observe previous loads. More interesting are loads that are satisfied by buffer forwarding. When we execute a buffer forwarded load e_l to $\mathbf{x}$, we do not observe *any* new event, since the load was not able to reach and synchronize with the memory. Instead we save the load event with the buffer entry from which it read its value. When that entry is updated to memory, by the update event $\mathsf{upd}_{\mathsf{ld}}(e_l)$, we move e_l to the set of loads that have been observed by $\mathcal{M}(\mathbf{x})$. By this scheme the load event e_l becomes available for observation by precisely the update events which succeed $\mathsf{upd}_{\mathsf{ld}}(e_l)$. In the remainder of this section we will make this intuition formal.

4.1 Instantiating *Source-DPOR* for chronological traces

We are now ready to show an instantiation of the *Source-DPOR* algorithm of [1] using chronological traces. In Fig. 11, we give the main DPOR algorithm **TSO-Source-DPOR**, explained in this section. The algorithm makes a call to the auxiliary algorithm **TSO-post** in order to compute the next configuration according to the TSO semantics, compute the corresponding chronological trace, and identify events which race with each other. The algorithm **TSO-post** is given in Fig. 12, and explained in Sect. 4.2.

The DPOR algorithm takes three parameters: the current execution τ, the current extended state $(\mathbb{L}, \mathbb{M}, \mathcal{C}, \mathcal{B}, \mathcal{M})$ and a sleep set *Sleep* of threads which are currently blocked from being executed. The algorithm recursively explores executions which are continuations of τ. On line 1 we pick a thread p that can be executed in the current state, and which is not in the sleep set. The next instruction of p will be the first continuation of τ which is explored. As races are discovered between events during the exploration, new alternative continuations will be added to the set `backtrack`. Such alternatives will be explored in subsequent iterations through the loop on lines 3–30.

For each event e that is added to τ in one iteration of the loop, three main steps are performed: On line 7, the configuration is updated to reflect the effect of executing e, as well as the new edges in the happens-before relation $\rightarrow_{\tau}^{\mathsf{ct}}$. At the same time, we compute the set `cnf` of earlier events which race with e. On lines 8–16, we add new branches to `backtrack`, corresponding to the races that have been detected and collected in the set `cnf`. On lines 17–27 we update the sleep set in order to unblock those threads whose next instruction races with e.

The first step is handled by the auxiliary algorithm in Fig. 12. It is explained in Sect. 4.2 below.

In the next step, on lines 8–16, we add new branches to the set `backtrack`, corresponding to each of the races that have been collected in `cnf`. For each event e_c which is in a race with e, we want to add an alternative branch, where e_c is delayed, allowing the possibility for e to execute before e_c. Therefore, at line 9, we identify the position in τ where e_c was executed, and the sub-executions τ_0 and τ_1 preceding and succeeding e_c. We then identify the set `I` of events in τ_1 which may be the first executed event after τ_0 in a hypothetical other execution where e executes before e_c. We make certain at lines 12–15 that at least one of the events in `I` is represented as an alternative branch to explore after τ_0.

```
global backtrack $= \lambda\tau.\varnothing : \mathsf{Event}^* \mapsto 2^{\mathsf{TID}\cup\mathsf{AuxTID}}$
TSO-Source-DPOR($\tau$,$(\mathbb{L}, \mathbb{M}, \mathcal{C}, \mathcal{B}, \mathcal{M})$,$Sleep$)
 1: if($\exists p \in (\mathsf{TID} \cup \mathsf{AuxTID}) \setminus Sleep.$ $p$ is enabled){
 2:   backtrack($\tau$) := $\{p\}$;
 3:   while($\exists p \in$ backtrack($\tau$) $\setminus Sleep$){
 4:     op := the next instruction of $p$ in $(\mathbb{L}, \mathbb{M})$;
 5:     i := 1 + the number of events by $p$ in $\tau$;
 6:     $e$ := $(p, \mathsf{op}, \mathsf{i})$; // The new event
 7:     $((\mathbb{L}', \mathbb{M}', \mathcal{C}', \mathcal{B}', \mathcal{M}'),$cnf) := TSO-post($(\mathbb{L}, \mathbb{M}, \mathcal{C}, \mathcal{B}, \mathcal{M})$,$e$);
        // Add branches to backtrack for each detected race in cnf
 8:     for($e_c \in$ cnf){
 9:       $\tau_0 \cdot e_c \cdot \tau_1$ := $\tau$; // Identify position of $e_c$ in $\tau$
10:       $\tau_1'$ := $\tau_1 \cdot e$ with all events $e'$ s.t. $\mathcal{C}'(e_c) \le \mathcal{C}'(e')$ removed;
          // Set I to the set of initial events in $\tau_1'$
11:       I := $\{e_i \in \tau_1' \mid \not\exists e_i' \in \tau_1'.\mathcal{C}'(e_i') \le \mathcal{C}'(e_i)\}$;
12:       if(I $\cap$ backtrack($\tau_0$) $= \varnothing$){ // No initial in the backtrack set
13:         $e_i$ := pick an element in I;
14:         backtrack($\tau_0$) := backtrack($\tau_0$) $\cup \{\mathsf{tid}(e_i)\}$;
15:       }
16:     }
        // Remove racing threads from the sleep set.
17:     $Sleep'$ := $Sleep$;
18:     if($\exists$x. op accesses x){ // $e$ is a memory access
19:       $Sleep'$ := $\varnothing$;
20:       for($q \in Sleep$){
21:         op' := the next instruction of $q$ in $(\mathbb{L}', \mathbb{M}')$;
22:         y := the memory location accessed by op';
23:         if(y $\ne$ x or (both op and op' are loads) or ($q = \mathsf{upd}(p)$) or
               (op' is a load and $\mathcal{B}(q)$ contains a letter of the form $(\mathbf{y}, \_, \_)$)){
24:           $Sleep'$ := $Sleep' \cup \{q\}$;
25:         }
26:       }
27:     }
28:     TSO-Source-DPOR($\tau \cdot e$,$(\mathbb{L}', \mathbb{M}', \mathcal{C}', \mathcal{B}', \mathcal{M}')$,$Sleep'$);
29:     $Sleep$ := $Sleep \cup \{p\}$;
30:   }
31: }
```

Fig. 11 The source-DPOR algorithm of [1], adapted to TSO using chronological traces. The initial call is **TSO-Source-DPOR**(ε, $(\mathbb{L}_0, \mathbb{M}_0, \lambda p.\lambda q.0, \lambda p.\varepsilon, \lambda \mathbf{x}(\bot, \emptyset))$, $\emptyset$), where $(\mathbb{L}_0, \mathbb{M}_0)$ is an initial TSO configuration

When we explore a new branch, like the ones introduced above, we use a *sleep set* to ensure that events that are supposed to be delayed are delayed for sufficiently long. Thus when exploration of a new branch like the one above starts, the thread identifier of the previously racing event (e_c) is inserted into the sleep set *Sleep* (on line 29) and the event may not be executed until the thread is removed from that set. A thread should be removed from the sleep set when the execution has proceeded such that executing its next event would have a different effect than previously explored. This happens as soon as an event is executed that would be in a race with the sleeping thread's next event. Therefore, on lines 17–27, we identify which of the sleeping threads race with e, and remove them from *Sleep*.

Example exploration Recall the program given in Fig. 5. In the example of Fig. 8 in the preliminaries, we gave a high-level explanation of how DPOR with chronological traces

```
TSO-post($(\mathbb{L},\mathbb{M},\mathcal{C},\mathcal{B},\mathcal{M})$,$e$)
(p, op, i) := e;
($\mathbb{R},\mathbb{B}$) := $\mathbb{L}(p)$;
$\mathbb{R}_{\mathsf{pc}}$ := advance the program counter of $p$ in $\mathbb{R}$;
cnf := $\varnothing$; // Events racing with $e$
$\mathsf{c}_p$ := $\mathcal{C}(p)[p \hookleftarrow \mathrm{i}]$; // Tick the clock for $p$
$(\mathbb{L}',\mathbb{M}',\mathcal{C}',\mathcal{B}',\mathcal{M}')$ := $(\mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}},\mathbb{B})],\mathbb{M},\mathcal{C}[e,p \hookleftarrow \mathsf{c}_p],\mathcal{B},\mathcal{M})$;
// Case split based on instruction type
if($\exists \mathbf{x},\$r$.op = store: $\mathbf{x}$:=$\$r$){ // Store
    $\mathbb{L}'$ := $\mathbb{L}'[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}},\mathbb{B}\cdot(\mathbf{x},\mathbb{R}(\$r)))]$; // Add new store to buffer
    $\mathcal{B}'$ := $\mathcal{B}[p \hookleftarrow \mathcal{B}(p)\cdot(\mathbf{x},e,\perp)]$; // Aux info for new buffer entry
}else if($\exists \mathbf{x}$.op = $\mathsf{u}(\mathbf{x})$){ // Update
    $(\mathbf{x},v)\cdot\mathbb{B}'$ := $\mathbb{B}$;
    $\mathbb{L}'$ := $\mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}},\mathbb{B}')]$; // Remove store from buffer
    $\mathbb{M}'$ := $\mathbb{M}[\mathbf{x} \hookleftarrow v]$; // Update memory
    $(\mathbf{x},e_s,e_r)\cdot\mathsf{b}_p$ := $\mathcal{B}(p)$; // Get aux info for store
    $\mathcal{B}'$ := $\mathcal{B}[p \hookleftarrow \mathsf{b}_p]$; // Remove store from aux buffer
    $(e_u,\mathrm{L})$ := $\mathcal{M}(\mathbf{x})$;
    if($e_r = \perp$){
        $\mathcal{M}'$ := $\mathcal{M}[\mathbf{x} \hookleftarrow (e,\mathrm{L})]$;
    }else{
        $\mathcal{M}'$ := $\mathcal{M}[\mathbf{x} \hookleftarrow (e,\mathrm{L}\oplus e_r)]$;
    }
    $\mathcal{C}'$ := $\mathcal{C}[p,e \hookleftarrow \mathsf{c}_p \sqcup \mathcal{C}(e_s) \sqcup \mathcal{C}(e_u) \sqcup \bigsqcup_{e_l\in\mathrm{L}\text{ s.t. }\mathsf{upd}(\mathsf{tid}(e_l))\neq p}\mathcal{C}(e_l)]$;
    cnf := $\left\{e' \in \mathrm{L}\cup\{e_u\} \,\middle|\, \begin{array}{l} e' \neq \perp \text{ and } \mathcal{C}(e') \not\leq \mathsf{c}_p \sqcup \mathcal{C}(e_s) \text{ and} \\ \mathsf{tid}(e') \neq p \text{ and } \mathsf{upd}(\mathsf{tid}(e')) \neq p \end{array}\right\}$;
}else if($\exists \mathbf{x},\$r$.op = load: $\$r$:=$\mathbf{x}$ and $\mathbb{B}(\mathbf{x}) = \perp$){ // Load from memory
    $\mathbb{L}'$ := $\mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}[\$r \hookleftarrow \mathbb{M}(\mathbf{x})],\mathbb{B})]$;
    $(e_u,\mathrm{L})$ := $\mathcal{M}(\mathbf{x})$;
    $\mathcal{M}'$ := $\mathcal{M}[\mathbf{x} \hookleftarrow (e_u,\mathrm{L}\oplus e)]$;
    if($e_u \neq \perp$ and $\mathsf{tid}(e_u) \neq \mathsf{upd}(p)$){
        $\mathcal{C}'$ := $\mathcal{C}[e,p \hookleftarrow \mathsf{c}_p \sqcup \mathcal{C}(e_u)]$;
        cnf := $\{e_u\}$;
    }
}else if($\exists \mathbf{x},\$r,v$.op = load: $\$r$:=$\mathbf{x}$ and $\mathbb{B}(\mathbf{x}) = v \neq \perp$){ // Load from buffer
    $\mathbb{L}'$ := $\mathbb{L}[p \hookleftarrow (\mathbb{R}_{\mathsf{pc}}[\$r \hookleftarrow \mathbb{B}(\mathbf{x})],\mathbb{B})]$;
    $\mathsf{b}_p\cdot(\mathbf{x},e_s,e_r)\cdot\mathsf{b}_p{}'$ := $\mathcal{B}(p)$ where
        $\mathsf{b}_p{}'$ contains no elements of the form $(\mathbf{x},\_,\_)$;
    $\mathcal{B}'$ := $\mathcal{B}[p \hookleftarrow \mathsf{b}_p\cdot(\mathbf{x},e_s,e)\cdot\mathsf{b}_p{}']$;
}else if(op = fence){ // Fence
    $\mathcal{C}'$ := $\mathcal{C}[e,p \hookleftarrow \mathsf{c}_p \sqcup \mathcal{C}(\mathsf{upd}(p))]$;
}else{ // Some local instruction
    $(\mathbb{L}',\mathbb{M}',\mathcal{C}',\mathcal{B}',\mathcal{M}')$ := perform appropriate local modifications;
}
return $((\mathbb{L}',\mathbb{M}',\mathcal{C}',\mathcal{B}',\mathcal{M}'),\mathtt{cnf})$;
```

Fig. 12 An algorithm which computes the extended configuration reached by executing the event e from the extended configuration $(\mathbb{L},\mathbb{M},\mathcal{C},\mathcal{B},\mathcal{M})$. The algorithm returns the new configuration, as well as a set `cnf`, which contains the earlier events which race with e

would proceed to explore that program. In Fig. 13, we revisit that exploration, and point to how it is achieved by the algorithm given in Fig. 11.

At first, the sleep set is empty, as well as the backtrack set of the current execution $\tau = \varepsilon$. At this point the threads p and q are enabled, but not the auxiliary threads $\mathsf{upd}(p)$ and

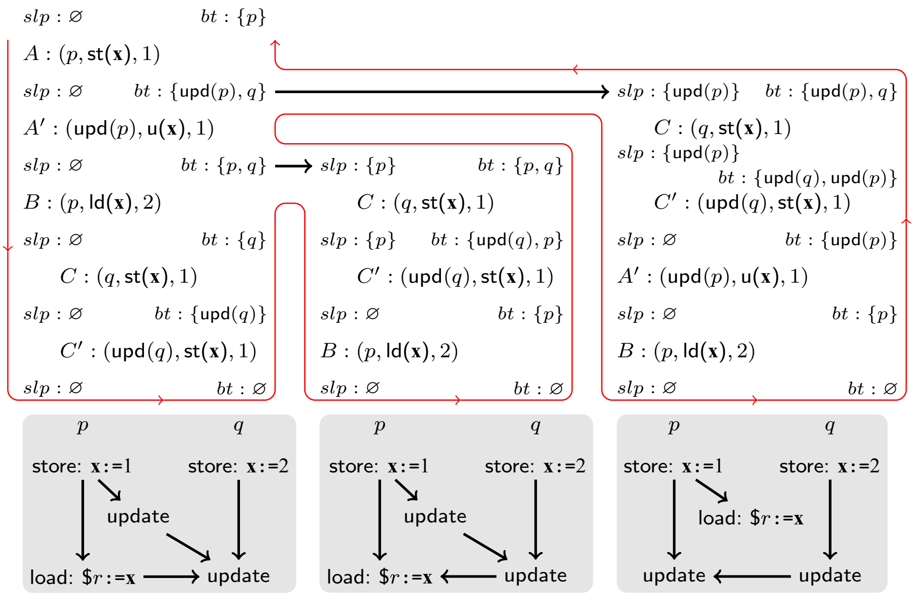


Fig. 13 An exploration by **TSO-Source-DPOR** of the program in Fig. 5

upd(q). On line 1 of the algorithm, we pick the enabled thread p, and proceed to insert p into the backtrack set of τ and execute the first instruction A of p. In Fig. 13, we see the execution of A at the top left. The slp : $\emptyset$ and bt : $\{p\}$ above A indicate that the sleep set at this point is empty, and the backtrack set is the singleton set containing p.

We continue arbitrarily scheduling the instructions of the enabled threads in the sequence A', B, C, C', down along the left-most column of Fig. 13. This gives us the first execution.

As we execute each event, the call to **TSO-post** on line 7 identifies the earlier events with which the current event has a conflict. During the first execution, two conflicts are identified: When the update C' is executed, we will have $\mathcal{M}(\mathbf{x}) = (A', \{B\})$, and we will find a conflict from A' to C' and one from B to C'. Hence on line 7 when C' is executed, `cnf` will be assigned $\{A', B\}$. When we enter the loop on lines 8-16, with $e_c = A'$ we will first split the execution $\tau = AA'BC$ into $\tau_0 = A$ and $\tau_1 = BC$ on line 9. Then we will identify the events in $\tau_1 \cdot e = BCC'$ which are initial. Notice that C precedes C' in the happens-before order, but no events in BCC' precede B or C. Hence we assign $\mathtt{I} = \{B, C\}$. On lines 12–15 we make certain that either B or C is in the backtrack set corresponding to the position immediately before A' was executed. In this case we choose to insert tid(C) $= q$ into $\mathtt{backtrack}(A)$. Similarly in the next iteration of the loop on lines 8–16, we insert q into $\mathtt{backtrack}(AA')$.

As the first execution has been completely explored, the algorithm now starts to backtrack. After C' has been explored, on the bottom left in Fig. 13, its thread upd(q) is added to the sleep set on line 29. Since the only thread in the backtrack set (upd(q)) is also in the sleep set, the loop on lines 3–30 terminates, and the call to **TSO-Source-DPOR** returns. The call to **TSO-Source-DPOR** which executed C immediately returns in the same way. In the call which executed B however, the backtrack set now contains one additional thread q which should be explored. Therefore, the algorithm takes an extra lap in the loop on lines 3-30, this time exploring the instruction C of q. This new branch is illustrated in Fig. 13 as the middle column. Notice that p is present in the sleep set, which prevents us from scheduling the load B until some other conflicting event has been executed.

As the update C' is executed in the middle execution, we again identify that it has a conflict with the earlier update A'. Again we find that C is an initial event along the execution between A' and C'. But since $\mathsf{tid}(C) = q$ is already present in the backtrack set where A' is executed, we do not need to insert it again (i.e., $\mathtt{I} \cap \mathtt{backtrack}(\tau_0) \neq \emptyset$ on line 12 in the algorithm).

We also identify that the update C' conflicts with the event B which is the next event of the thread p which is in the sleep set. Therefore, we remove p from the sleep set on lines 17–27 in the algorithm.

This leaves the algorithm free to schedule B as the next and last event of the middle execution. As we execute the load B, we have $\mathcal{M}(\mathbf{x}) = (C', \emptyset)$, we therefore detect a conflict from the update C' to the load B. As a result, $\mathsf{tid}(B) = p$ is inserted into the backtrack set $\mathtt{backtrack}(AA'C)$ immediately before C'.

As before, we now start to backtrack. When we reach the call to **TSO-Source-DPOR** where C' was executed, we find both $\mathsf{upd}(q)$ and p in the backtrack set. However, both are also in the sleep set, and so the call returns without starting a new branch. The next call returns similarly, and we return to the left-most column in Fig. 13. On the call which executed A', we find the thread q in the backtrack set but not in the sleep set. We then start the new branch corresponding to the right-most column in Fig. 13.

The thread $\mathsf{upd}(p)$ corresponding to the update A' is added to the sleep set and cannot be scheduled until the conflicting update C' has been executed. This effectively ensures that the order of the two updates is reversed in the last execution. As A' is executed, we identify that it is in conflict with the earlier C', and therefore add $\mathsf{tid}(A') = \mathsf{upd}(p)$ to the backtrack set $\mathtt{backtrack}(AC)$. When the last event, the load B, is executed, we have $\mathcal{M}(\mathbf{x}) = (A', \emptyset)$. Since the update A' originates in the same thread as the load B, there is no conflict from A' to B, and so we do not update any backtrack sets.

When backtracking after the last execution, at no point do we have a thread which is present in the backtrack set but not in the sleep set. Therefore, no new branches are initiated, and the algorithm terminates.

4.2 Computing the next configuration and happens-before relation

We call the algorithm $\mathtt{TSO\text{-}post}((\mathbb{L}, \mathbb{M}, \mathcal{C}, \mathcal{B}, \mathcal{M}), e)$, shown in Fig. 12, to compute the extended configuration which is reached by executing the event e from the configuration $(\mathbb{L}, \mathbb{M}, \mathcal{C}, \mathcal{B}, \mathcal{M})$. The algorithm performs a case split based on the type of instruction that is being executed. We will here pay some attention to the case of updates, and leave the other cases undescribed, since they are similar. The update case is covered on lines 10–23. First, on lines 11–13 we remove the oldest store from the store buffer, and update the memory, as described in the TSO semantics above. On the next two lines, we remove the corresponding entry from the buffer $\mathcal{B}$ in the extended configuration. At the same time, we take note that the event e_s is the store corresponding to this update, and that e_r is the latest load to which this store has been buffer forwarded. On lines 16–21, we update the information about the memory location $\mathbf{x}$ in the extended configuration. We change it such that e is recorded as the latest update for $\mathbf{x}$. Furthermore, if the update e has been buffer forwarded to any load e_r, then that load is recorded as the latest load of $\mathbf{x}$ by p. By recording e_r in $\mathcal{M}(\mathbf{x})$ at this point, rather than at the point when the load itself was executed, we ensure that the load is recorded as racing with updates which succeed e, but not with "hidden" updates which precede e. Next, on line 22, we assign a new vector clock to both the thread p and the event e. The new vector clock is the pointwise maximum of the vector clocks of all events that precede e in the $\xrightarrow{\mathsf{ct}}_{\tau}$ order. The new clock includes $\mathtt{c}_p$, which captures the program order $\xrightarrow{\mathsf{po}}_{\tau}$, and $\mathcal{C}(e_s)$ which captures the relation $\xrightarrow{\mathsf{su}}_{\tau}$ to e from the store e_s from which it originates. It includes $\mathcal{C}(e_u)$

which captures the relation $\rightarrow_{\tau}^{\mathsf{uu}}$ to e from the last previous update to $\mathbf{x}$, and it includes $\mathcal{C}(e_l)$ for all previous loads e_l to $\mathbf{x}$, capturing the conflict relation $\rightarrow_{\tau}^{\mathsf{cf\text{-}ct}}$. Finally, on line 23, we record the previous memory accesses which are in a race with e, i.e., the events originating in different threads, and which immediately precede e in $\rightarrow_{\tau}^{\mathsf{uu}} \cup \rightarrow_{\tau}^{\mathsf{src\text{-}ct}} \cup \rightarrow_{\tau}^{\mathsf{cf\text{-}ct}}$. Hence, we select events from the update e_u and the loads in $\mathtt{L}$ which have not already been ordered in $\rightarrow_{\tau}^{\mathsf{ct}}$ with e or its corresponding store e_s, and which have different thread identifiers.

4.3 Correctness of the DPOR algorithms

We state the following theorem of correctness for DPOR applied to chronological traces.

Theorem 2 (Correctness of DPOR algorithms) *The* Source-DPOR *and* Optimal-DPOR *algorithms of [1], 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.*

Proof of Theorem 2 The proof of Theorem 2 mainly uses the correctness of *Source-DPOR*, which is proven in [1]. More precisely, in [1] it is proven that *Source-DPOR* is correct whenever it is based on an assignment of happens-before relations to executions, which is *valid*. An assignment of happens-before relations $\rightarrow_{\tau}$ to executions τ is valid if it satisfies the following natural properties (from [1]).

1. $\rightarrow_{\tau}$ is a partial order on the events in τ, which is included in $<_{\tau}$,
2. the events of each thread are totally ordered by $\rightarrow_{\tau}$,
3. if τ' is a prefix of τ, then $\rightarrow_{\tau}$ and $\rightarrow_{\tau'}$ are the same on τ'.
4. the assignment of happens-before relations to executions partitions the set of executions into equivalence classes; i.e., if τ' is a linearization of the happens-before relation on τ, then τ' is assigned the same happens-before relation as τ; we use $\simeq$ to denote the corresponding equivalence relation,
5. whenever τ and τ' are equivalent then they end up in the same global program state,
6. for any sequences τ, τ' and τ'', such that $\tau \cdot \tau''$ is an execution, we have $\tau \simeq \tau'$ if and only if $\tau \cdot \tau'' \simeq \tau' \cdot \tau''$, and
7. if $\tau \cdot (p, i, j)$ is an execution, whose last event is performed by thread p, and q, r are different threads, such that (p, i, j) would "happen before" a subsequent event by r but not a subsequent event by q, then (p, i, j) would also "happen before" (r, i'', j'') in the execution $\tau \cdot (p, i, j) \cdot (q, i', j') \cdot (r, i'', j'')$.

A consequence of these definitions is that that if e and e' are two consecutive events in τ with $e \not\rightarrow \tau e'$, then e and e' can be swapped without affecting the (global) state after the two events.

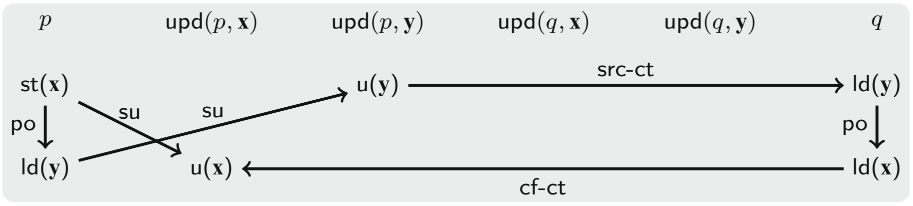
The theorem can now be proven by establishing that the happens-before assignment induced by chronological traces is valid. Conditions 1, 2, 3, and 6 follow straight-forwardly from definitions Condition 4 follows by observing that changing the order between non-related events does not affect the definition of the chronological trace. Condition 5 follows by observing that the chronological trace captures all dependences that are needed for determining which values are read and written by loads and stores. Finally, Condition 7 follows by noting that an arrow between (p, i, j) and (r, i'', j'') in a chronological trace cannot be removed by inserting an event that is independent with p. This concludes the proof of Theorem 2.

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

(a)

$(p, \mathsf{st}(\mathbf{x}), 1)$
$(p, \mathsf{st}(\mathbf{y}), 2)$
$(\mathsf{upd}(p, \mathbf{y}), \mathsf{u}(\mathbf{y}), 1)$
$(q, \mathsf{ld}(\mathbf{y}), 1)$
$(q, \mathsf{ld}(\mathbf{x}), 2)$
$(\mathsf{upd}(p, \mathbf{x}), \mathsf{u}(\mathbf{x}), 1)$

(b)


p
$\mathsf{upd}(p, \mathbf{x})$
$\mathsf{upd}(p, \mathbf{y})$
$\mathsf{upd}(q, \mathbf{x})$
$\mathsf{upd}(q, \mathbf{y})$
q
$\mathsf{st}(\mathbf{x})$
$\mathsf{u}(\mathbf{y})$
src-ct
$\mathsf{ld}(\mathbf{y})$
po
su
su
po
$\mathsf{ld}(\mathbf{y})$
$\mathsf{u}(\mathbf{x})$
$\mathsf{ld}(\mathbf{x})$
cf-ct

(c)

Fig. 14 A behavior allowed under PSO but not under TSO. **a** The mp idiom. Possible in the final configuration under PSO: $\$r = 1$, $\$s = 0$. **b** An execution τ where $\$r = 1$, $\$s = 0$ holds in the final configuration. **c** The chronological trace of τ under PSO

5 Adaptation for PSO

In this section, we show how our techniques can be adapted to the PSO memory model with minor changes. Before we see how to apply our methods to it, we give an informal description of the PSO memory model.

5.1 PSO semantics

PSO is a strictly more relaxed model than TSO. As described previously, TSO allows reordering of stores with subsequent loads. PSO allows the same reordering, but also allows the reordering of stores with subsequent stores to different memory locations.

This behavior can be modelled by an operational semantics similar to the one described in Sect. 3 for TSO, but where each thread has a separate store buffer for each memory location. Each store buffer is FIFO-ordered, so stores to the same memory location by the same thread cannot be reordered. But there is no order maintained between stores in different buffers, so stores by the same thread to different locations may update in reversed order.

In Fig. 14a we give an example of a program where PSO allows more behaviors than TSO. The execution in Fig. 14b shows how the stores by p to $\mathbf{x}$ and $\mathbf{y}$ update to memory in reversed order. This allows the thread q to read first $\mathbf{y} = 1$ then $\mathbf{x} = 0$, which would be impossible both under SC and TSO.

In the operational semantics for PSO we introduce one auxiliary thread $\mathsf{upd}(p, \mathbf{x})$ for each pair of a thread p and a memory location $\mathbf{x}$. Each such auxiliary thread is responsible for the updates to $\mathbf{x}$ by p, similarly to how $\mathsf{upd}(p)$ under TSO is responsible for all updates of p.

5.2 Chronological traces for PSO

The adaptation of chronological traces to PSO is straightforward. The following simple adjustment suffices: Since stores from the same thread p to different memory locations $\mathbf{x}$ and

$\mathbf{y}$ are updated by different auxiliary threads $\mathsf{upd}\ (p,\mathbf{x})$ and $\mathsf{upd}\ (p,\mathbf{y})$, there is no program order edge between the update events for different memory locations under PSO.

Formally, we reuse the definition of chronological traces for TSO, from Sect. 3, with some minor changes:

We need to reformulate the definition of $\mathsf{upd}_{\mathsf{st}}$ to reflect that there are now multiple store buffers per thread: For an execution τ and an event $e = (p, \mathsf{st}(\mathbf{x}), j)$ in τ, let k be the number of events $e_w = (p', \mathsf{st}(\mathbf{x}), j')$ in τ such that $p' = p$ and $j' \leq j$. Then $\mathsf{upd}_{\mathsf{st}}(e) = (\mathsf{upd}(p, \mathbf{x}), \mathsf{u}(\mathbf{x}), k)$ if there is such an event in τ. Otherwise $\mathsf{upd}_{\mathsf{st}}(e) = e^{\infty}$. This new definition of $\mathsf{upd}_{\mathsf{st}}$ replaces the old one in the definition of chronological traces for PSO.

We can then define chronological traces for PSO in the same way as for TSO, except that the definition of $\rightarrow_{\tau}^{\mathsf{uf}}$ needs to be reformulated as follows:

For two events $e = (p, i, j)$ and $e' = (p', i', j')$ we say that $e \rightarrow_{\tau}^{\mathsf{uf\text{-}pso}} e'$ iff $i = \mathsf{u}(\mathbf{x})$ for some $\mathbf{x}$, and $i' = \mathsf{fence}$ and $p = \mathsf{upd}(p', \mathbf{x})$ and $e <_{\tau} e'$ and there is no event $e'' = (p, \mathsf{u}(\mathbf{x}), j'')$ such that $e <_{\tau} e'' <_{\tau} e'$. I.e., under PSO, we put an edge in $\rightarrow_{\tau}^{\mathsf{uf\text{-}pso}}$ to the fence from the last preceding update by the thread for each memory location, rather than as under TSO only from the single last preceding update by the thread to any memory location.

A chronological trace for PSO is illustrated in Fig. 14c. Notice that there is no program order edge from $(\mathsf{upd}(p, \mathbf{x}), \mathsf{u}(\mathbf{x}), 1)$ to $(\mathsf{upd}(p, \mathbf{y}), \mathsf{u}(\mathbf{y}), 1)$. Had there been one, the trace would be cyclic.

6 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 [26] 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 Sect. 5 in Abdulla et al. [1], 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.

7 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. Other programs, 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 Sect. 8.

Since both SMC and BMC require that all runs of the analyzed program terminate within some finite bound, we apply loop bounding when analyzing the benchmarks. The bound is indicated in the LB column of Table 1. Furthermore, all benchmarks are data-deterministic, since this is a requirement for SMC, as mentioned earlier.

All experiments were run on a machine equipped with a 3 GHz Intel i7 processor and 6 GB RAM running 64-bit Linux. We used 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.

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. 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.

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

	Fence	LB	CBMC			goto-instrument			Nidhugg		
			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

Table 1 continued

	Fence	LB	CBMC			goto-instrument			Nidhugg		
			SC	TSO	PSO	SC	TSO	PSO	SC	TSO	PSO
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**

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

8 Related work

To the best of our knowledge, our work, together with the work by Zhang et al. [41] are the first to apply stateless model checking techniques to the setting of relaxed memory models; see e.g. [1] for a recent survey of related work on stateless model checking and dynamic partial order reduction techniques. The work by Zhang et al. [41] was developed independently and concurrently with the work presented in this paper, and shares many similarities with it.

There have been many previous works dedicated to the verification and checking of programs running under RMM (e.g., [3,7–11,21,23,25,40]). 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., [10,11,25]). There are also a number of efforts to design bounded model checking techniques for programs under RMM (e.g., [9,40]) which encode the verification problem in SAT.

The two closest works to ours are those presented in [5,6]. 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.

There has also been some recent work on SAT-directed stateless model checking [20], including consideration of RMM [14]. The main idea is to encode some key parts of the concurrent program into a SAT formula and hand it over to a general purpose SMT solver to produce additional interleavings. For the most relevant tool, SATCheck [14], the authors claim that this approach scales better with the length of program execution, basing their evaluation on increasing the length of the traces by increasing the number of iterations of a small program core. We were not able to evaluate SATCheck on our own benchmark set, as the tool is currently at a prototype level and requires preprocessing by an expert user in order to handle arbitrary programs. Nevertheless, experimentation on one of our benchmark programs (`dekker.c`) confirms that performance of Nidhugg and SATCheck are similar for small programs.

9 Concluding remarks

We have presented a 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 plan to extend Nidhugg to more memory models such as Power, ARM, and the C/C++ memory model. This will require adapting the definition of chronological traces to those models in order to guarantee the one-to-one correspondence with Shasha–Snir traces.

References

1. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL, pp. 373–384. ACM (2014)
2. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Tools and algorithms for the construction and analysis of systems, pp. 353–367. Springer, Heidelberg (2015)
3. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS, vol. 7214 of LNCS, pp. 204–219. Springer (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: ESOP, vol. 7792 of LNCS, pp. 512–532. Springer (2013)
6. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV, vol. 8044 of LNCS, pp. 141–157. Springer (2013)
7. Alglave, J., Maranget, L.: Stability in weak memory models. In: CAV, vol. 6806 of LNCS, pp. 50–66. Springer (2011)
8. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP, vol. 7792 of LNCS, pp. 533–553. Springer (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: CAV, vol. 5123 of LNCS, pp. 107–120. Springer (2008)
11. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: TACAS, pp. 11–25. LNCS 6605, Springer (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. Demsky, B., Lam, P.: SATCheck: SAT-directed stateless model checking for SC and TSO. In: OOPSLA (2015)
15. Dijkstra, E.W.: Cooperating sequential processes. Springer, Heidelberg (2002)
16. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121. ACM (2005)
17. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem, vol. 1032 of LNCS. Springer (1996)
18. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL, pp. 174–186. ACM Press (1997)
19. Godefroid, P.: Software model checking: the VeriSoft approach. Formal Methods Syst. Des. **26**(2), 77–101 (2005)
20. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. In: PLDI, pp. 165–174. New York, NY, USA, ACM (2015)
21. Krishnamurthy, A., Yelick, K.A.: Analyses and optimizations for shared address space programs. J. Parallel Distrib. Comput. **38**(2), 130–144 (1996)
22. Lauterburg, S., Karmani, R., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: FASE, pp 308–322. LNCS 6013 (2010)
23. Lee, J., Padua, D.A.: Hiding relaxed memory consistency with a compiler. IEEE Trans. Comput. **50**(8), 824–833 (2001)
24. Lei, Y., Carver, R.: Reachability testing of concurrent programs. IEEE Trans. Softw. Eng. **32**(6), 382–403 (2006)

25. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI, pp. 429–440. ACM (2012)
26. The LLVM compiler infrastructure. http://llvm.org
27. Mazurkiewicz, A.: Trace theory. In: Advances in Petri Nets (1986)
28. 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)
29. Park, S., Dill, D.L: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: SPAA, pp. 34–41. ACM (1995)
30. Peled, D.: All from one, one for all, on model-checking using representatives. In: CAV, vol. 697 of LNCS, pp. 409–423. Springer (1993)
31. Peterson, G., Stickel, M.: Myths about the mutal exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
32. Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: ACSD, IEEE (2012)
33. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Haifa Verification Conference, pp. 166–182. LNCS 4383 (2007)
34. 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. ACM **53**(7), 89–97 (2010)
35. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. **10**(2), 282–312 (1988)
36. SPARC International, Inc. The SPARC Architecture Manual Version 9 (1994)
37. Tasharofi, S. et al.: TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In: FMOODS/FORTE, pp. 219–234. LNCS 7273 (2012)
38. Valmari, A.: Stubborn sets for reduced state space generation. In: Advances in Petri Nets, vol. 483 of LNCS, pp. 491–515. Springer (1989)
39. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: ICSE, pp. 221–230. ACM (2011)
40. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: IPDPS, IEEE (2004)
41. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation, pp. 250–259. ACM (2015)