The Reads-From Equivalence for the TSO and PSO Memory Models

TRUC LAM BUI*, Comenius University, Slovakia
KRISHNENDU CHATTERJEE, IST Austria, Austria
TUSHAR GAUTAM*, IIT Bombay, India
ANDREAS PAVLOGIANNIS, Aarhus University, Denmark
VIKTOR TOMAN, IST Austria, Austria

The verification of concurrent programs remains an open challenge due to the non-determinism in inter-process communication. One recurring algorithmic problem in this challenge is the consistency verification of concurrent executions. In particular, consistency verification under a reads-from map allows to compute the *reads-from (RF) equivalence* between concurrent traces, with direct applications to areas such as Stateless Model Checking (SMC). Importantly, the RF equivalence was recently shown to be coarser than the standard Mazurkiewicz equivalence, leading to impressive scalability improvements for SMC under SC (sequential consistency). However, for the *relaxed memory* models of TSO and PSO (total/partial store order), the algorithmic problem of deciding the RF equivalence, as well as its impact on SMC, has been elusive.

In this work we solve the algorithmic problem of consistency verification for the TSO and PSO memory models given a reads-from map, denoted VTSO-rf and VPSO-rf, respectively. For an execution of n events over k threads and d variables, we establish novel bounds that scale as n^{k+1} for TSO and as $n^{k+1} \cdot \min(n^{k^2}, 2^{k \cdot d})$ for PSO. Moreover, based on our solution to these problems, we develop an SMC algorithm under TSO and PSO that uses the RF equivalence. The algorithm is *exploration-optimal*, in the sense that it is guaranteed to explore each class of the RF partitioning exactly once, and spends polynomial time per class when k is bounded. Finally, we implement all our algorithms in the SMC tool Nidhugg, and perform a large number of experiments over benchmarks from existing literature. Our experimental results show that our algorithms for VTSO-rf and VPSO-rf provide significant scalability improvements over standard alternatives. Moreover, when used for SMC, the RF partitioning is often much coarser than the standard Shasha–Snir partitioning for TSO/PSO, which yields a significant speedup in the model checking task.

CCS Concepts: • **Theory of computation → Verification by model checking**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: concurrency, relaxed memory models, execution-consistency verification, stateless model checking

ACM Reference Format:
Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. 2021. The Reads-From Equivalence for the TSO and PSO Memory Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 164 (October 2021), 30 pages. https://doi.org/10.1145/3485541

*Work done while the author was an intern at IST Austria.

Authors' addresses: Truc Lam Bui, Comenius University, Mlynská dolina, Bratislava, 842 48, Slovakia, bujtuclam@gmail.com; Krishnendu Chatterjee, IST Austria, Am Campus 1, Klosterneuburg, 3400, Austria, krishnendu.chatterjee@ist.ac.at; Tushar Gautam, IIT Bombay, Main Gate Rd, IIT Area, Powai, Mumbai, 400076, India, tushargautam.gautam@gmail.com; Andreas Pavlogiannis, Aarhus University, Nordre Ringgade 1, Aarhus, 8000, Denmark, pavlogiannis@cs.au.dk; Viktor Toman, IST Austria, Am Campus 1, Klosterneuburg, 3400, Austria, viktor.toman@ist.ac.at.

2475-1421/2021/10-ART164
https://doi.org/10.1145/3485541

1 INTRODUCTION

The formal analysis of concurrent programs is a key problem in program analysis and verification. Scheduling non-determinism makes programs both hard to write correctly, and to analyze formally, as both the programmer and the model checker need to account for all possible communication patterns among threads. This non-determinism incurs an exponential blow-up in the state space of the program, which in turn yields a significant computational cost on the verification task.

Traditional verification has focused on concurrent programs adhering to sequential consistency [Lamport 1979]. Programs operating under relaxed memory semantics exhibit additional behavior compared to sequential consistency. This makes it exceptionally hard to reason about correctness, as, besides scheduling subtleties, the formal reasoning needs to account for buffer/caching mechanisms. Two of the most standard operational relaxed memory models in the literature are *Total Store Order* (TSO) and *Partial Store Order* (PSO) [Adve and Gharachorloo 1996; Alglave 2010; Alglave et al. 2017; Owens et al. 2009; Sewell et al. 2010; SPARC International 1994].

On the operational level, both models introduce subtle mechanisms via which write operations become visible to the shared memory and thus to the whole system. Under TSO, every thread is equipped with its own buffer. Every write to a shared variable is pushed into the buffer, and thus remains hidden from the other threads. The buffer is flushed non-deterministically to the shared memory, at which point the writes become visible to the other threads. The semantics under PSO are even more involved, as now every thread has one buffer *per shared variable*, and non-determinism now governs not only when a thread flushes its buffers, but also which buffers are flushed.

Thread_1	Thread_2
1. $w(x)$	1. $w'(y)$
2. $\mathrm{r}(y)$	2. $\mathrm{r}'(x)$

Thread_1	Thread_2
1. $w(x)$	1. $\mathrm{r}(y)$
2. $w'(y)$	2. $\mathrm{r}'(x)$

Fig. 1. A TSO example (left) and a PSO example (right).

To illustrate the intricacies under TSO and PSO, consider the examples in Figure 1. On the left, under SC, in every execution at least one of $\mathrm{r}(y)$ and $\mathrm{r}'(x)$ will observe the corresponding $w'(y)$ and $w(x)$. Under TSO, however, the write events may become visible on the shared memory only after the read events have executed, and hence both write events go unobserved. Executions under PSO are even more involved, see Figure 1 right. Under either SC or TSO, if $\mathrm{r}(y)$ observes $w'(y)$, then $\mathrm{r}'(x)$ must observe $w(x)$, as $w(x)$ becomes visible on the shared memory before $w'(y)$. Under PSO, however, there is a single local buffer for each variable. Hence the order in which $w(x)$ and $w'(y)$ become visible in the shared memory can be reversed, allowing $\mathrm{r}(y)$ to observe $w'(y)$ while $\mathrm{r}'(x)$ does not observe $w(x)$.

The great challenge in verification under relaxed memory is to systematically, yet efficiently, explore all such extra behaviors of the system, i.e., account for the additional non-determinism that comes from the buffers. In this work we tackle this challenge for two verification tasks under TSO and PSO, namely, for verifying the consistency of executions, and for stateless model checking.

Verifying execution consistency with a reads-from function. One of the most basic problems for a given memory model is the verification of the consistency of program executions with respect to the given model [Chini and Saivasan 2020]. The input is a set of thread executions, where each execution performs operations accessing the shared memory. The task is to verify whether the thread executions can be interleaved to a concurrent execution, which has the property that every read observes a specific value written by some write [Gibbons and Korach 1997]. The problem is of

foundational importance to concurrency, and has been studied heavily under SC [Cain and Lipasti 2002; Chen et al. 2009; Hu et al. 2012].

The input is often enhanced with a *reads-from (RF) map*, which further specifies for each read access the write access that the former should observe. Under sequential consistency, the corresponding problem VSC-rf was shown to be *NP*-hard in the landmark work of Gibbons and Korach [1997], while it was recently shown W[1]-hard [Mathur et al. 2020]. The problem lies at the heart of many verification tasks in concurrency, such as dynamic analyses [Kini et al. 2017; Mathur et al. 2020, 2021; Pavlogiannis 2019; Roemer et al. 2020; Smaragdakis et al. 2012], linearizability and transactional consistency [Biswas and Enea 2019; Herlihy and Wing 1990], as well as SMC [Abdulla et al. 2019; Chalupa et al. 2017; Kokologiannakis et al. 2019b].

Executions under relaxed memory. The natural extension of verifying execution consistency with an RF map is from SC to relaxed memory models such as TSO and PSO, we denote the respective problems by VTSO-rf and VPSO-rf. Given the importance of VSC-rf for SC, and the success in establishing both upper and lower bounds, the complexity of VTSO-rf and VPSO-rf is a very natural question and of equal importance. The verification problem is known to be *NP*-hard for most memory models [Furbach et al. 2015], including TSO and PSO, however, no other bounds are known. Some heuristics have been developed for VTSO-rf [Manovit and Hangal 2006; Zennou et al. 2019], while other works study TSO executions that are also sequentially consistent [Bouajjani et al. 2013, 2011].

Stateless Model Checking. The most standard solution to the space-explosion problem is *stateless model checking* [Godefroid 1996]. Stateless model-checking methods typically explore traces rather than states of the analyzed program. The depth-first nature of the exploration enables it to be both systematic and memory-efficient, by storing only a few traces at any given time. Stateless model-checking techniques have been employed successfully in several well-established tools, e.g., VeriSoft [Godefroid 1997, 2005] and CHESS [Madan Musuvathi 2007].

As there are exponentially many interleavings, a trace-based exploration typically has to explore exponentially many traces, which is intractable in practice. One standard approach is the partitioning of the trace space into equivalence classes, and then attempting to explore every class via a single representative trace. The most successful adoption of this technique is in *dynamic partial order reduction (DPOR)* techniques [Clarke et al. 1999; Flanagan and Godefroid 2005; Godefroid 1996; Peled 1993]. The great advantage of DPOR is that it handles indirect memory accesses precisely without introducing spurious interleavings. The foundation underpinning DPOR is the famous Mazurkiewicz equivalence, which constructs equivalence classes based on the order in which traces execute conflicting memory access events. This idea has led to a rich body of work, with improvements using symbolic techniques [Kahlon et al. 2009], context-sensitivity [Albert et al. 2017], unfoldings [Rodríguez et al. 2015], effective lock handling [Kokologiannakis et al. 2019a], and others [Albert et al. 2018; Aronis et al. 2018; Chatterjee et al. 2019]. The work of Abdulla et al. [2014] developed an SMC algorithm that is exploration-optimal for the Mazurkiewicz equivalence, in the sense that it explores each class of the underlying partitioning exactly once. Finally, techniques based on SAT/SMT solvers have been used to construct even coarser partitionings [Demsky and Lam 2015; Huang 2015; Huang and Huang 2017].

The reads-from equivalence for SMC. A new direction of SMC techniques has been recently developed using the *reads-from (RF)* equivalence to partition the trace space. The key principle is to classify traces as equivalent based on whether read accesses observe the same write accesses. The idea was initially explored for acyclic communication topologies [Chalupa et al. 2017], and has been recently extended to all topologies [Abdulla et al. 2019]. As the RF partitioning is guaranteed

to be (even exponentially) coarser than the Mazurkiewicz partitioning, SMC based on RF has shown remarkable scalability potential [Abdulla et al. 2019, 2018; Kokologiannakis et al. 2019b; Kokologiannakis and Vafeiadis 2020]. The key technical component for SMC using RF is the verification of execution consistency, as presented in the previous section. The success of SMC using RF under SC has thus rested upon new efficient methods for the problem VSC-rf.

SMC under relaxed memory. The SMC literature has taken up the challenge of model checking concurrent programs under relaxed memory. Extensions to SMC for TSO/PSO have been considered by Zhang et al. [2015] using shadow threads to model memory buffers, as well as by Abdulla et al. [2015] using chronological traces to represent the Shasha–Snir notion of trace under relaxed memory [Shasha and Snir 1988]. Chronological/Shasha–Snir traces are the generalization of Mazurkiewicz traces to TSO/PSO. Further extensions have also been made to other memory models, namely by Abdulla et al. [2018] for the release-acquire fragment of C++11, Kokologiannakis et al. [2017, 2019b] for the RC11 model [Lahav et al. 2017], and Kokologiannakis and Vafeiadis [2020] for the IMM model [Podkopaev et al. 2019], but notably none for TSO and PSO using the RF equivalence. Given the advantages of the RF equivalence for SMC under SC [Abdulla et al. 2019], release-acquire [Abdulla et al. 2018], RC11 [Kokologiannakis et al. 2019b] and IMM [Kokologiannakis and Vafeiadis 2020], a very natural standing question is whether RF can be used for effective SMC under TSO and PSO. Here we tackle this challenge.

1.1 Our Contributions

Here we outline the main results of our work. We refer to Section 3 for a formal presentation.

Verifying execution consistency for TSO and PSO. Our first set of results and the main contribution of this paper is on the problems VTSO-rf and VPSO-rf for verifying TSO- and PSO-consistent executions, respectively. Consider an input to the corresponding problem that consists of k threads and n operations, where each thread executes write and read operations, as well as *fence* operations that flush each thread-local buffer to the main memory. Our results are as follows.

(1) We present an algorithm that solves VTSO-rf in $O(k \cdot n^{k+1})$ time. The case of VSC-rf is solvable in $O(k \cdot n^k)$ time [Abdulla et al. 2019; Biswas and Enea 2019; Mathur et al. 2020]. Although for TSO there are k additional buffers, our result shows that the complexity is only minorly impacted by an additional factor n, as opposed to n^k.
(2) We present an algorithm that solves VPSO-rf in $O(k \cdot n^{k+1} \cdot \min(n^{k \cdot (k-1)}, 2^{k \cdot d}))$ time, where d is the number of variables. Note that even though there are $k \cdot d$ buffers, one of our two bounds is independent of d and thus yields polynomial time when the number of threads is bounded. Moreover, our bound collapses to $O(k \cdot n^{k+1})$ when there are no fences, and hence this case is no more difficult that VTSO-rf.

Stateless model checking for TSO and PSO using the reads-from equivalence (RF). Our second contribution is an algorithm RF-SMC for SMC under TSO and PSO using the RF equivalence. The algorithm is based on the reads-from algorithm for SC [Abdulla et al. 2019] and uses our solutions to VTSO-rf and VPSO-rf for visiting each class of the respective partitioning. Moreover, RF-SMC is *exploration-optimal*, in the sense that it explores only maximal traces and further it is guaranteed to explore each class of the RF partitioning exactly once. For the complexity statements, let k be the total number of threads and n be the number of events of the longest trace. The time spent by RF-SMC per class of the RF partitioning is

(1) $n^{O(k)}$ time, for the case of TSO, and
(2) $n^{O(k^2)}$ time, for the case of PSO.

Note that the time complexity per class is polynomial in n when k is bounded.

Implementation and experiments. We have implemented RF-SMC in the stateless model checker Nidhugg [Abdulla et al. 2015], and performed an evaluation on an extensive set of benchmarks from the recent literature. Our results show that our algorithms for VTSO-rf and VPSO-rf provide significant scalability improvements over standard alternatives, often by orders of magnitude. Moreover, when used for SMC, the RF partitioning is often much coarser than the standard Shasha–Snir partitioning for TSO/PSO, which yields a significant speedup in the model checking task.

2 PRELIMINARIES

General notation. Given a natural number $i \geq 1$, we let $[i]$ be the set $\{1, 2, \ldots, i\}$. Given a map $f : X \to Y$, we let $\mathrm{dom}(f) = X$ and $\mathrm{img}(f) = Y$ denote the domain and image of f, respectively. We represent maps f as sets of tuples $\{(x, f(x))\}_x$. Given two maps f_1, f_2 over the same domain X, we write $f_1 = f_2$ if for every $x \in X$ we have $f_1(x) = f_2(x)$. Given a set $X' \subset X$, we denote by $f|X'$ the restriction of f to X'. A binary relation $\sim$ on a set X is an *equivalence* iff $\sim$ is reflexive, symmetric and transitive. We denote by $X/\!\sim$ the *quotient* (i.e., the set of all equivalence classes) of X under $\sim$.

2.1 Concurrent Model under TSO/PSO

Here we describe the computational model of concurrent programs with shared memory under the Total Store Order (TSO) and Partial Store Order (PSO) memory models. We follow a standard exposition, similarly to Abdulla et al. [2015]; Huang and Huang [2016]. We first describe TSO and then extend our description to PSO.

Concurrent program with Total Store Order. We consider a concurrent program $\mathscr{P} = \{\mathsf{thr}_i\}_{i=1}^{k}$ of k threads. The threads communicate over a shared memory $\mathcal{G}$ of global variables. Each thread additionally owns a *store buffer*, which is a FIFO queue for storing updates of variables to the shared memory. Threads execute *events* of the following types.

(1) A *buffer-write event* wB enqueues into the local store buffer an update that wants to write a value v to a global variable $x \in \mathcal{G}$.
(2) A *read event* r reads the value v of a global variable $x \in \mathcal{G}$. The value v is the value of the most recent local buffer-write event, if one still exists in the buffer, otherwise v is the value of x in the shared memory.

Additionally, whenever a store buffer of some thread is nonempty, the respective thread can execute the following.

(3) A *memory-write event* wM that dequeues the oldest update from the local buffer and performs the corresponding write-update on the shared memory.

Threads can also flush their local buffers into the memory using fences.

(4) A *fence event* fnc blocks the corresponding thread until its store buffer is empty.

Finally, threads can execute local events that are not modeled explicitly, as usual. We refer to all non-memory-write events as *thread events*. Following the typical setting of stateless model checking [Abdulla et al. 2014, 2015; Chalupa et al. 2017; Flanagan and Godefroid 2005], each thread of the program $\mathscr{P}$ is deterministic, and further $\mathscr{P}$ is bounded, meaning that all executions of $\mathscr{P}$ are finite and the number of events of $\mathscr{P}$'s longest execution is a parameter of the input.

Given an event e, we denote by $\mathsf{thr}(e)$ its thread and by $\mathrm{var}(e)$ its global variable. We denote by $\mathcal{E}$ the set of all events, by $\mathcal{R}$ the set of read events, by $\mathcal{W}^B$ the set of buffer-write events, by $\mathcal{W}^M$ the set of memory-write events, and by $\mathcal{F}$ the set of fence events. Given a buffer-write event $\mathrm{wB} \in \mathcal{W}^B$ and its corresponding memory-write $\mathrm{wM} \in \mathcal{W}^M$, we let $\mathbf{w} = (\mathrm{wB}, \mathrm{wM})$ be the two-phase write event, and we denote $\mathsf{thr}(\mathbf{w}) = \mathsf{thr}(\mathrm{wB}) = \mathsf{thr}(\mathrm{wM})$ and $\mathrm{var}(\mathbf{w}) = \mathrm{var}(\mathrm{wB}) = \mathrm{var}(\mathrm{wM})$. We denote by $\mathcal{W}$ the set of all such two-phase write events. Given two events $e_1, e_2 \in \mathcal{R} \cup \mathcal{W}^M$, we say that

they *conflict*, denoted $e_1 \bowtie e_2$, if they access the same global variable and at least one of them is a memory-write event.

Proper event sets. Given a set of events $X \subseteq \mathcal{E}$, we write $\mathcal{R}(X) = X \cap \mathcal{R}$ for the set of read events of X, and similarly $\mathcal{W}^B(X) = X \cap \mathcal{W}^B$ and $\mathcal{W}^M(X) = X \cap \mathcal{W}^M$ for the buffer-write and memory-write events of X, respectively. We also denote by $\mathcal{L}(X) = X \setminus \mathcal{W}^M(X)$ the thread events (i.e., the non-memory-write events) of X. We write $\mathcal{W}(X) = (X \times X) \cap \mathcal{W}$ for the set of two-phase write events in X. We call X *proper* if $\mathrm{wB} \in X$ iff $\mathrm{wM} \in X$ for each $(\mathrm{wB}, \mathrm{wM}) \in \mathcal{W}$. Finally, given a set of events $X \subseteq \mathcal{E}$ and a thread thr, we denote by X_{thr} and $X_{\neq \mathsf{thr}}$ the events of thr, and the events of all other threads in X, respectively.

Sequences and Traces. Given a sequence of events $\tau = e_1, \ldots, e_j$, we denote by $\mathcal{E}(\tau)$ the set of events that appear in τ. We further denote $\mathcal{R}(\tau) = \mathcal{R}(\mathcal{E}(\tau))$, $\mathcal{W}^B(\tau) = \mathcal{W}^B(\mathcal{E}(\tau))$, $\mathcal{W}^M(\tau) = \mathcal{W}^M(\mathcal{E}(\tau))$, and $\mathcal{W}(\tau) = \mathcal{W}(\mathcal{E}(\tau))$. Finally we denote by ϵ an empty sequence.

Given a sequence τ and two events $e_1, e_2 \in \mathcal{E}(\tau)$, we write $e_1 <_\tau e_2$ when e_1 appears before e_2 in τ, and $e_1 \leq_\tau e_2$ to denote that $e_1 <_\tau e_2$ or $e_1 = e_2$. Given a sequence τ and a set of events A, we denote by $\tau|A$ the *projection* of τ on A, which is the unique sub-sequence of τ that contains all events of $A \cap \mathcal{E}(\tau)$, and only those. Given a sequence τ and an event $e \in \mathcal{E}(\tau)$, we denote by $\mathrm{pre}_\tau(e)$ the prefix up until and including e, formally $\tau|\{e' \in \mathcal{E}(\tau) \mid e' \leq_\tau e\}$. Given two sequences τ_1 and τ_2, we denote by $\tau_1 \circ \tau_2$ the sequence that results in appending τ_2 after τ_1.

A (concrete, concurrent) *trace* is a sequence of events σ that corresponds to a concrete valid execution of $\mathscr{P}$ under standard semantics [Shasha and Snir 1988]. We let $\mathsf{enabled}(\sigma)$ be the set of enabled events after σ is executed, and call σ *maximal* if $\mathsf{enabled}(\sigma) = \emptyset$. A concrete *local trace* ρ is a sequence of thread events of the same thread.

Reads-from functions. Given a proper event set $X \subseteq \mathcal{E}$, a *reads-from function* over X is a function that maps each read event of X to some two-phase write event of X accessing the same global variable. Formally, $\mathsf{RF}\colon \mathcal{R}(X) \to \mathcal{W}(X)$, where $\mathrm{var}(\mathrm{r}) = \mathrm{var}(\mathsf{RF}(\mathrm{r}))$ for all $\mathrm{r} \in \mathcal{R}(X)$. Given a buffer-write event wB (resp. a memory-write event wM), we write $\mathsf{RF}(\mathrm{r}) = (\mathrm{wB}, \_)$ (resp. $\mathsf{RF}(\mathrm{r}) = (\_, \mathrm{wM})$) to denote that $\mathsf{RF}(\mathrm{r})$ is a two-phase write for which wB (resp. wM) is the corresponding buffer-write (resp. memory-write) event.

Given a sequence of events τ where the set $\mathcal{E}(\tau)$ is proper, we define the *reads-from function* of τ, denoted $\mathsf{RF}_\tau\colon \mathcal{R}(\tau) \to \mathcal{W}(\tau)$, as follows. Given a read event $\mathrm{r} \in \mathcal{R}(\tau)$, consider the set $\mathrm{Upd} = \{(\mathrm{wB}, \mathrm{wM}) \in (\mathcal{W}(\tau))_{\mathsf{thr}(\mathrm{r})} \mid \mathrm{wM} \bowtie \mathrm{r},\ \mathrm{wB} <_\tau \mathrm{r} <_\tau \mathrm{wM}\}$. Then, $\mathsf{RF}_\tau(\mathrm{r}) = (\mathrm{wB}', \mathrm{wM}')$, where one of the two cases happens:

- $\mathrm{Upd} \neq \emptyset$, and $(\mathrm{wB}', \mathrm{wM}') \in \mathrm{Upd}$ is the latest in τ, i.e., for each $(\mathrm{wB}'', \mathrm{wM}'') \in \mathrm{Upd}$ we have $\mathrm{wB}'' \leq_\tau \mathrm{wB}'$.
- $\mathrm{Upd} = \emptyset$, and $\mathrm{wM}' \in \mathcal{W}^M(\tau)$, $\mathrm{wM}' \bowtie \mathrm{r}$, $\mathrm{wM}' <_\tau \mathrm{r}$ is the latest memory-write (of any thread) conflicting with r and occurring before r in τ, i.e., for each $\mathrm{wM}'' \in \mathcal{W}^M(\tau)$ such that $\mathrm{wM}'' \bowtie \mathrm{r}$ and $\mathrm{wM}'' <_\tau \mathrm{r}$, we have $\mathrm{wM}'' \leq_\tau \mathrm{wM}'$.

Notice how relaxed memory comes into play in the above definition, as $\mathsf{RF}_\tau(\mathrm{r})$ does not record which of the two above cases actually happened.

Partial Store Order and Sequential Consistency. The memory model of Partial Store Order (PSO) is more relaxed than TSO. On the operational level, each thread is equipped with a store buffer for each global variable, rather than a single buffer for all global variables. Then, at any point during execution, a thread can non-deterministically dequeue and perform the oldest update from any of its nonempty store buffers. The notions of events, traces and reads-from functions remain the same for PSO as defined for TSO. The Sequential Consistency (SC) memory model can

be simply thought of as a model where each thread flushes its buffer immediately after a write event, e.g., by using a fence.

Concurrent program semantics. The semantics of $\mathscr{P}$ are defined by means of a transition system over a state space of global states. A global state consists of (i) a memory function that maps every global variable to a value, (ii) a local state for each thread, which contains the values of the local variables of the thread, and (iii) a local state for each store buffer, which captures the contents of the queue. We consider the standard setting with the TSO/PSO memory model, and refer to Abdulla et al. [2015] for formal details. As usual in stateless model checking, we focus on concurrent programs with acyclic state spaces.

Reads-from trace partitioning. Given a concurrent program $\mathscr{P}$ and a memory model $\mathcal{M} \in \{\text{SC}, \text{TSO}, \text{PSO}\}$, we denote by $\mathcal{T}_{\mathcal{M}}$ the set of maximal traces of the program $\mathscr{P}$ under the respective memory model. We call two traces σ_1 and σ_2 *reads-from equivalent* if $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$ and $\text{RF}_{\sigma_1} = \text{RF}_{\sigma_2}$. The corresponding *reads-from equivalence* $\sim_{\text{RF}}$ partitions the trace space into equivalence classes $\mathcal{T}_{\mathcal{M}}/\sim_{\text{RF}}$ and we call this the *reads-from partitioning* (or RF partitioning). Traces in the same class of the RF partitioning visit the same set of local states in each thread, and thus the RF partitioning is a sound partitioning for local state reachability [Abdulla et al. 2019; Chalupa et al. 2017; Kokologiannakis et al. 2019b].

2.2 Partial Orders

Here we present relevant notation around partial orders.

Partial orders. Given a set of events $X \subseteq \mathcal{E}$, a *(strict) partial order* P over X is an irreflexive, antisymmetric and transitive relation over X (i.e., $<_P \subseteq X \times X$). Given two events $e_1, e_2 \in X$, we write $e_1 \leq_P e_2$ to denote that $e_1 <_P e_2$ or $e_1 = e_2$. Two distinct events $e_1, e_2 \in X$ are *unordered* by P, denoted $e_1 \parallel_P e_2$, if neither $e_1 <_P e_2$ nor $e_2 <_P e_1$, and *ordered* (denoted $e_1 \not\parallel_P e_2$) otherwise. Given a set $Y \subseteq X$, we denote by $P|Y$ the *projection* of P on the set Y, where for every pair of events $e_1, e_2 \in Y$, we have that $e_1 <_{P|Y} e_2$ iff $e_1 <_P e_2$. Given two partial orders P and Q over a common set X, we say that Q *refines* P, denoted by $Q \sqsubseteq P$, if for every pair of events $e_1, e_2 \in X$, if $e_1 <_P e_2$ then $e_1 <_Q e_2$. A *linearization* of P is a total order that refines P.

Lower sets. Given a pair (X, P), where X is a set of events and P is a partial order over X, a *lower set* of (X, P) is a set $Y \subseteq X$ such that for every event $e_1 \in Y$ and event $e_2 \in X$ such that $e_2 \leq_P e_1$, we have $e_2 \in Y$.

The program order PO. The *program order* PO of $\mathscr{P}$ is a partial order $<_{\text{PO}} \subseteq \mathcal{E} \times \mathcal{E}$ that defines a fixed order between some pairs of events of the same thread. Given any (concrete) trace σ and thread thr, the buffer-writes, reads, and fences of thr that appear in σ are fully ordered in PO the same way as they are ordered in σ. Further, for each thread thr, the program order PO satisfies the following conditions:

- $\text{wB} <_{\text{PO}} \text{wM}$ for each $(\text{wB}, \text{wM}) \in \mathcal{W}_{\text{thr}}$.
- $\text{wB} <_{\text{PO}} \text{fnc}$ iff $\text{wM} <_{\text{PO}} \text{fnc}$ for each $(\text{wB}, \text{wM}) \in \mathcal{W}_{\text{thr}}$ and fence event $\text{fnc} \in \mathcal{F}_{\text{thr}}$.
- $\text{wB}_1 <_{\text{PO}} \text{wB}_2$ iff $\text{wM}_1 <_{\text{PO}} \text{wM}_2$ for each $(\text{wB}_i, \text{wM}_i) \in \mathcal{W}_{\text{thr}}, i \in \{1, 2\}$. In PSO, this condition is enforced only when $\text{var}((\text{wB}_1, \text{wM}_1)) = \text{var}((\text{wB}_2, \text{wM}_2))$.

A sequence τ is *well-formed* if it respects the program order, i.e., $\tau \sqsubseteq \text{PO}|\mathcal{E}(\tau)$. Naturally, every trace σ is well-formed, as it corresponds to a concrete valid program execution.

3 SUMMARY OF RESULTS

Here we present formally the main results of this paper. In later sections we present the details, algorithms and examples. The proofs appear in the appendix of Bui et al. [2021].

Verifying execution consistency for TSO and PSO. Our first set of results and the main contribution of this paper is on the problems VTSO-rf and VPSO-rf for verifying TSO- and PSO-consistent executions, respectively. The corresponding problem VSC-rf for Sequential Consistency (SC) was recently shown to be in polynomial time for a constant number of threads [Abdulla et al. 2019; Biswas and Enea 2019]. The solution for SC is obtained by essentially enumerating all the n^k possible lower sets of the program order (X, PO), where k is the number of threads, and hence yields a polynomial when $k = O(1)$. For TSO, the number of possible lower sets is $n^{2 \cdot k}$, since there are k threads and k buffers (one for each thread). For PSO, the number of possible lower sets is $n^{k \cdot (d+1)}$, where d is the number of variables, since there are k threads and $k \cdot d$ buffers (d buffers for each thread). Hence, following an approach similar to Abdulla et al. [2019]; Biswas and Enea [2019] would yield a running time of a polynomial with degree $2 \cdot k$ for TSO, and with degree $k \cdot (d+1)$ for PSO (thus the solution for PSO is not polynomial-time even when the number of threads is bounded). In this work we show that both problems can be solved significantly faster.

Theorem 3.1. VTSO-rf *for n events and k threads is solvable in $O(k \cdot n^{k+1})$ time.*

Theorem 3.2. VPSO-rf *for n events, k threads and d variables is solvable in $O(k \cdot n^{k+1} \cdot \min(n^{k \cdot (k-1)}, 2^{k \cdot d}))$. Moreover, if there are no fences, the problem is solvable in $O(k \cdot n^{k+1})$ time.*

Novelty. For TSO, Theorem 3.1 yields an improvement of order n^{k-1} compared to the naive $n^{2 \cdot k}$ bound. For PSO, perhaps surprisingly, the first upper-bound of Theorem 3.2 does not depend on the number of variables. Moreover, when there are no fences, the cost for PSO is the same as for TSO (with or without fences).

Stateless Model Checking for TSO and PSO. Our second result concerns stateless model checking (SMC) under TSO and PSO. We introduce an SMC algorithm RF-SMC that explores the RF partitioning in the TSO and PSO settings, as stated in the following theorem.

Theorem 3.3. *Consider a concurrent program $\mathscr{P}$ with k threads and d variables, under a memory model $\mathcal{M} \in \{\text{TSO}, \text{PSO}\}$ with trace space $\mathcal{T}_{\mathcal{M}}$ and n being the number of events of the longest trace in $\mathcal{T}_{\mathcal{M}}$.* RF-SMC *is a sound, complete and exploration-optimal algorithm for local state reachability in $\mathscr{P}$, i.e., it explores only maximal traces and visits each class of the RF partitioning exactly once. The time complexity is $O(\alpha \cdot |\mathcal{T}_{\mathcal{M}}/{\sim_{\text{RF}}}|)$, where*
(1) $\alpha = n^{O(k)}$ under $\mathcal{M} = \text{TSO}$, and
(2) $\alpha = n^{O(k^2)}$ under $\mathcal{M} = \text{PSO}$.

An algorithm with RF exploration-optimality in SC is presented by Abdulla et al. [2019]. Our RF-SMC algorithm generalizes the above approach to achieve RF exploration-optimality in the relaxed memory models TSO and PSO. Further, the time complexity of RF-SMC per class of RF partitioning is equal between PSO and TSO for programs with no fence instructions.

RF-SMC uses the verification algorithms developed in Theorem 3.1 and Theorem 3.2 as black-boxes to decide whether any specific class of the RF partitioning is TSO- or PSO-consistent, respectively. We remark that these theorems can potentially be used as black-boxes to other SMC algorithms that explore the RF partitioning (e.g., Chalupa et al. [2017]; Kokologiannakis et al. [2019b]; Kokologiannakis and Vafeiadis [2020]).

4 VERIFYING TSO AND PSO EXECUTIONS WITH A READS-FROM FUNCTION

In this section we tackle the verification problems VTSO-rf and VPSO-rf. In each case, the input is a pair (X, RF), where X is a proper set of events of $\mathscr{P}$, and $\text{RF} \colon \mathcal{R}(X) \to \mathcal{W}(X)$ is a reads-from function. The task is to decide whether there exists a trace σ that is a linearization of (X, PO) with $\text{RF}_\sigma = \text{RF}$, where RF_σ is wrt TSO/PSO memory semantics. In case such σ exists, we say that

(X, RF) is *realizable* and σ is its *witness* trace. We first define some relevant notation, and then establish upper bounds for VTSO-rf and VPSO-rf, i.e., Theorem 3.1 and Theorem 3.2.

Held variables. Given a trace σ and a memory-write $\mathrm{wM} \in \mathcal{W}^M(\sigma)$ present in the trace, we say that wM *holds* variable $x = \mathrm{var}(\mathrm{wM})$ in σ if the following hold.

(1) wM is the last memory-write event of σ on variable x.
(2) There exists a read event $\mathrm{r} \in X \setminus \mathcal{E}(\sigma)$ such that $\mathsf{RF}(\mathrm{r}) = (\_, \mathrm{wM})$.

We similarly say that the thread $\mathrm{thr}(\mathrm{wM})$ *holds* x in σ. Finally, a variable x is *held* in σ if it is held by some thread in σ. Intuitively, wM holds x until all reads that need to read-from wM get executed.

Witness prefixes. Throughout this section, we use the notion of witness prefixes. Formally, a *witness prefix* is a trace σ that can be extended to a trace σ^* that realizes (X, RF), under the respective memory model. Our algorithms for VTSO-rf and VPSO-rf operate by constructing traces σ such that if (X, RF) is realizable, then σ is a witness prefix that can be extended with the remaining events and finally realize (X, RF).

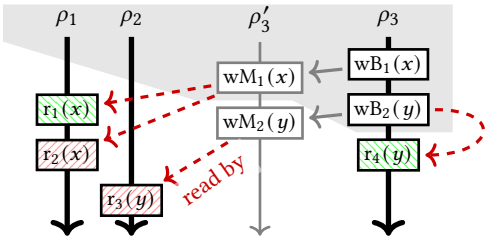
Throughout, we assume wlog that whenever $\mathsf{RF}(\mathrm{r}) = (\mathrm{wB}, \mathrm{wM})$ with $\mathrm{thr}(\mathrm{r}) = \mathrm{thr}(\mathrm{wB})$, then wB is the last buffer-write on $\mathrm{var}(\mathrm{wB})$ before r in their respective thread. Clearly, if this condition does not hold, then the corresponding pair (X, RF) is not realizable in TSO nor PSO.

4.1 Verifying TSO Executions

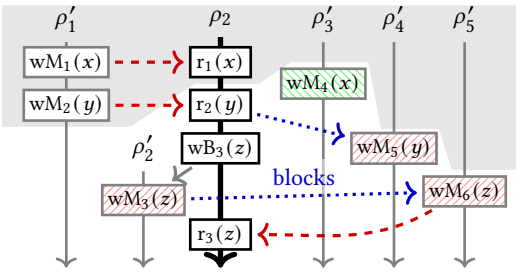
In this section we establish Theorem 3.1, i.e., we present an algorithm VerifyTSO that solves VTSO-rf in $O(k \cdot n^{k+1})$ time. The algorithm relies crucially on the notion of TSO-executable events, defined below. Throughout this section we consider fixed an instance (X, RF) of VTSO-rf, and all traces σ considered in this section are such that $\mathcal{E}(\sigma) \subseteq X$.

TSO-executable events. Consider a trace σ. An event $e \in X \setminus \mathcal{E}(\sigma)$ is TSO-*executable* (or executable for short) in σ if $\mathcal{E}(\sigma) \cup \{e\}$ is a lower set of (X, PO) and the following conditions hold.

(1) *If e is a read event* r, let $\mathsf{RF}(\mathrm{r}) = (\mathrm{wB}, \mathrm{wM})$. If $\mathrm{thr}(\mathrm{r}) \neq \mathrm{thr}(\mathrm{wM})$, then $\mathrm{wM} \in \sigma$.
(2) *If e is a memory-write event* wM then the following hold.
   (a) Variable $\mathrm{var}(\mathrm{wM})$ is not held in σ.
   (b) Let $\mathrm{r} \in \mathcal{R}(X)$ be an arbitrary read with $\mathsf{RF}(\mathrm{r}) = (\mathrm{wB}, \mathrm{wM})$ and $\mathrm{thr}(\mathrm{r}) \neq \mathrm{thr}(\mathrm{wM})$. For each two-phase write $(\mathrm{wB}', \mathrm{wM}')$ with $\mathrm{var}(\mathrm{r}) = \mathrm{var}(\mathrm{wB}')$ and $\mathrm{wB}' <_{\mathsf{PO}} \mathrm{r}$, we have $\mathrm{wM}' \in \sigma$.



(a) The reads r_1 and r_4 are TSO-executable. The read r_2 is not TSO-executable, because $\mathcal{E}(\sigma) \cup \{\mathrm{r}_2\}$ is not a lower set; neither is the read r_3, because $\mathsf{RF}(\mathrm{r}_3) = (\_, \mathrm{wM}_2)$ has not been executed yet.



(b) The memory-write wM_4 is TSO-executable. The other memory-writes are not; $\mathcal{E}(\sigma) \cup \{\mathrm{wM}_3\}$ is not a lower set, for wM_5 resp. wM_6, the blue dotted arrows show the events that they have to wait for, because of Item 2a resp. Item 2b (some buffer-writes are not displayed here for brevity).

Fig. 2. TSO-executability. The already executed events (i.e., $\mathcal{E}(\sigma)$) are in the gray zone, the remaining events are outside the gray zone. The buffer threads are gray and thin, the main threads are black and thick.

Intuitively, the conditions of executable events ensure that executing an event does not immediately create an invalid witness prefix. The lower-set condition ensures that the program order PO is respected. This is a sufficient condition for a buffer-write or a fence (in particular, for a fence this implies that the respective buffer is currently empty). The extra condition for a read ensures that its reads-from constraint is satisfied. The extra conditions for a memory-write prevent it from causing some reads-from constraint to become unsatisfiable.

Figure 2 illustrates the notion of TSO-executability on several examples. Observe that if σ is a valid trace, extending σ with an executable event (i.e., $\sigma \circ e$) also yields a valid trace that is well-formed, as, by definition, $\mathcal{E}(\sigma) \cup \{e\}$ is a lower set of (X, PO).

Algorithm VerifyTSO. We are now ready to describe our algorithm VerifyTSO for the problem VTSO-rf. At a high level, the algorithm enumerates all lower sets of $(\mathcal{W}^M(X), \text{PO})$ by constructing a trace σ with $\mathcal{W}^M(\sigma) = Y$ for every lower set Y of $(\mathcal{W}^M(X), \text{PO})$. The crux of the algorithm is to maintain the following. Each constructed trace σ is *maximal* in the set of thread events, among all witness prefixes with the same set of memory-writes. That is, for every witness prefix σ' with $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$, we have that $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$. Thus, the algorithm will only explore n^k traces, as opposed to $n^{2 \cdot k}$ from a naive enumeration of all lower sets of (X, PO).

The formal description of VerifyTSO is in Algorithm 1. The algorithm maintains a worklist $\mathcal{S}$ of prefixes and a set Done of already-explored lower sets of $(\mathcal{W}^M(X), \text{PO})$. In each iteration, the Line 4 loop makes the prefix maximal in the thread events, then Line 6 checks if we are done, otherwise the loop in Line 7 enumerates the executable memory-writes to extend the prefix with.

Algorithm 1: VerifyTSO

Input: An event set X and a reads-from function $\text{RF} : \mathcal{R}(X) \to \mathcal{W}(X)$
Output: A witness σ that realizes (X, RF) if (X, RF) is realizable under TSO, else $\bot$

1 $\mathcal{S} \leftarrow \{\epsilon\}$; Done $\leftarrow \{\emptyset\}$
2 **while** $\mathcal{S} \neq \emptyset$ **do**
3 Extract a trace σ from $\mathcal{S}$
4 **while** $\exists$ *thread event* e TSO-*executable in* σ **do**
5 $\sigma \leftarrow \sigma \circ e$ // Execute the thread event e
6 **if** $\mathcal{E}(\sigma) = X$ **then return** σ // Witness found
7 **foreach** *memory-write* wM *that is* TSO-*executable in* σ **do**
8 $\sigma_{\text{wM}} \leftarrow \sigma \circ \text{wM}$ // Execute wM
9 **if** $\nexists \sigma' \in$ Done *s.t.* $\mathcal{W}^M(\sigma_{\text{wM}}) = \mathcal{W}^M(\sigma')$ **then**
10 Insert σ_{wM} in $\mathcal{S}$ and in Done // Continue from σ_{wM}
11 **return** $\bot$

We now provide the insights behind the correctness of VerifyTSO. The correctness proof has two components: (i) soundness and (ii) completeness, which we present below.

Soundness. The soundness follows directly from the definition of TSO-executable events. In particular, when the algorithm extends a trace σ with a read r, where $\text{RF}(\text{r}) = (\text{wB}, \text{wM})$, the following hold.

(1) If $\text{thr}(\text{r}) \neq \text{thr}(\text{wB})$, then $\text{wM} \in \sigma$, since r became executable. Moreover, when wM appeared in σ, the variable $x = \text{var}(\text{wM})$ became held by wM, and remained held at least until the current step where r is executed. Hence, no other memory-write wM' with $\text{var}(\text{wM}') = x$ could have become executable in the meantime, to violate the observation of r. Moreover, r cannot read-from a local buffer write wB' with $\text{var}(\text{wB}') = x$, as by definition, when wM became executable, all

buffer-writes on x that are local to r and precede r must have been flushed to the main memory (i.e., wM′ must have also appeared in the trace).

(2) If thr(r) = thr(wB), then either wM has not appeared already in σ, in which case r reads-from wB from its local buffer, or wM has appeared in the trace and held its variable until r is executed, as in the previous item.

Completeness. Let σ' be an arbitrary witness prefix, VerifyTSO constructs a trace σ such that $\mathcal{W}^M(\sigma) = \mathcal{W}^M(\sigma')$ and $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$. This is because VerifyTSO constructs for every lower set Y of $(\mathcal{W}^M(X), \text{PO})$ a single representative trace σ with $\mathcal{W}^M(\sigma) = Y$. The key is to make σ *maximal* on the thread events, i.e., $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$ for any witness prefix σ' with $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$, and thus any memory-write wM that is executable in σ' is also executable in σ.

We now present the above insight in detail. Indeed, if wM is not executable in σ, one of the following holds. Let var(wM) = x.

(1) x is already held in σ. But since $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$ and any read of σ' also appears in σ, the variable x is also held in σ', thus wM is not executable in σ' either.

(2) There is a later read r $\notin \sigma$ that must read-from wM, but r is preceded by a local write (wB′, wM′) (i.e., wB′ $<_{\text{PO}}$ r) also on x, for which wM′ $\notin \sigma$. Since $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$, we have r $\notin \sigma'$, and as $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$, also wM′ $\notin \sigma'$. Thus wM is also not executable in σ'.

The final insight is on how the algorithm maintains the maximality invariant as it extends σ with new events. This holds because read events become executable as soon as their corresponding remote observation wM appears in the trace, and hence all such reads are executable for a given lower set of $(\mathcal{W}^M(X), \text{PO})$. All other thread events are executable without any further conditions. Figure 3 illustrates the intuition behind the maximality invariant. The following lemma states the formal correctness, which together with the complexity argument gives us Theorem 3.1.

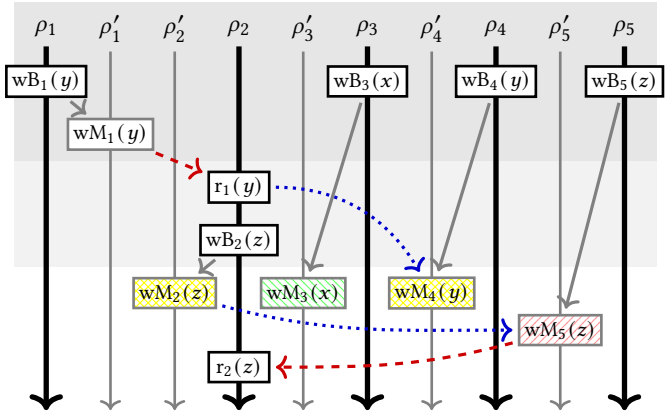


Fig. 3. VerifyTSO maximality invariant. The gray zone shows the events of some witness prefix σ'; the lighter gray shows the events of the corresponding trace σ, constructed by the algorithm, which is maximal on thread events. Yellow writes (wM_2 and wM_4) are those that are TSO-executable in σ but not in σ'. Green writes (wM_3) and red writes (wM_5) are TSO-executable and non TSO-executable, respectively.

Lemma 4.1. (X, RF) *is realizable under* TSO *iff* VerifyTSO *returns a trace* $\sigma \neq \epsilon$.

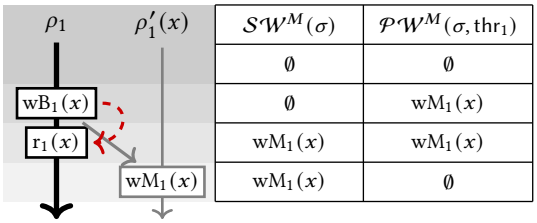
4.2 Verifying PSO Executions

In this section we show Theorem 3.2, i.e., we present an algorithm VerifyPSO that solves VPSO-rf in $O(k \cdot n^{k+1} \cdot \min(n^{k \cdot (k-1)}, 2^{k \cdot d}))$ time, while the bound becomes $O(k \cdot n^{k+1})$ when there are no fences. Similarly to the case of TSO, the algorithm relies on the notion of PSO-executable events, defined below. We first introduce some relevant notation that makes our presentation simpler.

Spurious and pending writes. Consider a trace σ with $\mathcal{E}(\sigma) \subseteq X$. A memory-write wM $\in \mathcal{W}^M(X)$ is called *spurious* in σ if the following conditions hold.

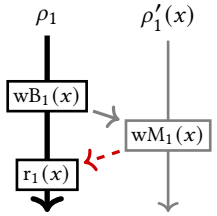
(1) There is no read $r \in \mathcal{R}(X) \setminus \sigma$ with $\mathrm{RF}(r) = (\_, wM)$
(informally, no remaining read wants to read-from wM).
(2) If $wM \in \sigma$, then for every read $r \in \sigma$ with $\mathrm{RF}_\sigma(r) = (\_, wM)$ we have $r <_\sigma wM$
(informally, reads in σ that read-from this write read it from the local buffer).

Note that if wM is a spurious memory-write in σ then wM is spurious in all extensions of σ. We denote by $\mathcal{SW}^M(\sigma)$ the set of memory-writes of σ that are spurious in σ. A memory-write wM is *pending* in σ if $wB \in \sigma$ and $wM \notin \sigma$, where wB is the corresponding buffer-write of wM. We denote by $\mathcal{PW}^M(\sigma, thr)$ the set of all pending memory-writes wM in σ with $thr(wM) = thr$. See Figure 4 for an intuitive illustration of spurious and pending memory-writes.



$\mathcal{SW}^M(\sigma)$	$\mathcal{PW}^M(\sigma, thr_1)$
$\emptyset$	$\emptyset$
$\emptyset$	$wM_1(x)$
$wM_1(x)$	$wM_1(x)$
$wM_1(x)$	$\emptyset$

(a) Linearization where wM_1 is spurious. The table shows the spurious and pending writes after each step.



(b) Linearization where wM_1 is *not* spurious; here $\mathrm{RF}_\sigma(r_1) = (\_, wM_1)$ and $wM_1 <_\sigma r_1$.

Fig. 4. Illustration of spurious and pending writes.

PSO-executable events. Similarly to the case of VTSO-rf, we define the notion of PSO-executable events (executable for short). An event $e \in X \setminus \mathcal{E}(\sigma)$ is PSO-*executable* in σ if the following conditions hold.

(1) *If e is a buffer-write or a memory-write*, then the same conditions apply as for TSO-executable.
(2) *If e is a fence* fnc, then every pending memory-write from thr(fnc) is PSO-executable in σ, and these memory-writes together with fnc and $\mathcal{E}(\sigma)$ form a lower set of (X, PO).
(3) *If e is a read* r, let $\mathrm{RF}(r) = (wB, wM)$. We have $wB \in \sigma$, and the following conditions.
   (a) if thr(r) = thr(wB), then $\mathcal{E}(\sigma) \cup \{r\}$ is a lower set of (X, PO).
   (b) if thr(r) ≠ thr(wB), then $\mathcal{E}(\sigma) \cup \{wM, r\}$ is a lower set of (X, PO) and further either $wM \in \sigma$ or wM is PSO-executable in σ.

Figure 5 illustrates several examples of PSO-(un)executable events. Similarly to the case of TSO, the PSO-executable conditions ensure that we do not execute events creating an invalid witness prefix. The executability conditions for PSO are different (e.g., there are extra conditions for a fence), since our approach for VPSO-rf fundamentally differs from the approach for VTSO-rf.

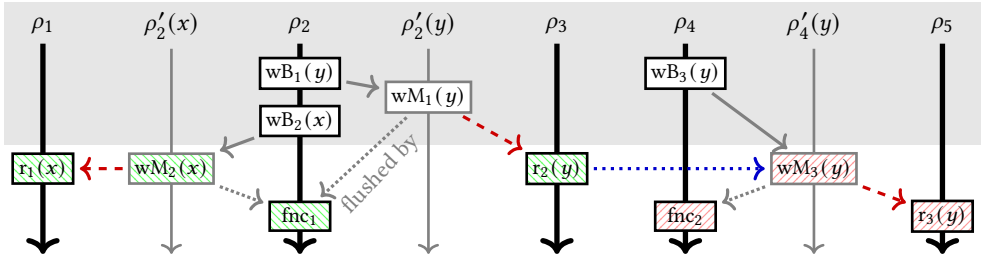


Fig. 5. PSO-executability. The green events are PSO-executable; the red events are not. The memory-write $wM_2(x)$ is executable, and thus so are $r_1(x)$ and fnc_1. The memory-write $wM_3(y)$ is not executable, as the variable y is held by $wM_1(y)$ until $r_2(y)$ is executed. Consequently, fnc_2 and $r_3(y)$ are not executable.

Fence maps. We define a *fence map* as a function $\text{FMap}_\sigma : \text{Threads} \times \text{Threads} \to [n]$ as follows. First, $\text{FMap}_\sigma(\text{thr}, \text{thr}) = 0$ for all $\text{thr} \in \text{Threads}$. In addition, if thr does not have a fence unexecuted in σ (i.e., a fence $\text{fnc} \in (X_{\text{thr}} \setminus \mathcal{E}(\sigma))$), then $\text{FMap}_\sigma(\text{thr}, \text{thr}') = 0$ for all $\text{thr}' \in \text{Threads}$. Otherwise, consider the set of all reads $A_{\text{thr},\text{thr}'}$ such that every $\text{r} \in A_{\text{thr},\text{thr}'}$ with $\text{RF}(\text{r}) = (\text{wB}, \text{wM})$ satisfies the following conditions.

(1) $\text{thr}(\text{r}) = \text{thr}'$ and $\text{r} \notin \sigma$.
(2) $\text{thr}(\text{wB}) \notin \{\text{thr}, \text{thr}'\}$, and $\text{var}(\text{r})$ is held by wM in σ, and there is a pending memory write wM' in σ with $\text{thr}(\text{wM}') = \text{thr}$ and $\text{var}(\text{wM}') = \text{var}(\text{r})$.

If $A_{\text{thr},\text{thr}'} = \emptyset$ then we let $\text{FMap}_\sigma(\text{thr}, \text{thr}') = 0$, otherwise $\text{FMap}_\sigma(\text{thr}, \text{thr}')$ is the largest index of a read in $A_{\text{thr},\text{thr}'}$. Given two traces σ_1, σ_2, $\text{FMap}_{\sigma_1} \leq \text{FMap}_{\sigma_2}$ denotes that $\text{FMap}_{\sigma_1}(\text{thr}, \text{thr}') \leq \text{FMap}_{\sigma_2}(\text{thr}, \text{thr}')$ for all $\text{thr}, \text{thr}' \in [k]$.

The intuition behind fence maps is as follows. Given a trace σ, the index $\text{FMap}_\sigma(\text{thr}, \text{thr}')$ points to the *latest* (wrt PO) read r of thr' that must be executed in any extension of σ before thr can execute its next fence. This occurs because the following hold in σ.

(1) The variable $\text{var}(\text{r})$ is held by the memory-write $\text{wM} \in \sigma$ with $\text{RF}(\text{r}) = (\_, \text{wM})$.
(2) Thread thr has executed some buffer-write $\text{wB}' \in \sigma$ with $\text{var}(\text{wB}') = \text{var}(\text{r}) = \text{var}(\text{wM})$, but the corresponding memory-write wM' has not yet been executed in σ. Hence, thr cannot flush its buffers in any extension of σ that does not contain r (as wM' will not become executable until r gets executed).

The following lemmas state two key monotonicity properties of fence maps.

Lemma 4.2. *Consider two witness prefixes σ_1, σ_2 such that $\sigma_2 = \sigma_1 \circ \text{wM}$ for some memory-write* wM *executable in σ_1. We have $\text{FMap}_{\sigma_1} \leq \text{FMap}_{\sigma_2}$. Moreover, if* wM *is a spurious memory-write in σ_1, then $\text{FMap}_{\sigma_1} = \text{FMap}_{\sigma_2}$.*

Lemma 4.3. *Consider two witness prefixes σ_1, σ_2 such that (i) $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$, (ii) $\text{FMap}_{\sigma_1} \leq \text{FMap}_{\sigma_2}$, and (iii) $\mathcal{W}^M(\sigma_1) \setminus \mathcal{SW}^M(\sigma_1) \subseteq \mathcal{W}^M(\sigma_2)$. Let $e \in \mathcal{L}(X)$ be a thread event that is executable in σ_i for each $i \in [2]$, and let $\sigma_i' = \sigma_i \circ e$, for each $i \in [2]$. Then $\text{FMap}_{\sigma_1'} \leq \text{FMap}_{\sigma_2'}$.*

Note that there exist in total at most $n^{k \cdot k}$ different fence maps. Further, the following lemma gives a bound on the number of different fence maps among witness prefixes that contain the same thread events.

Lemma 4.4. *Let d be the number of variables. There exist at most $2^{k \cdot d}$ distinct witness prefixes σ_1, σ_2 such that $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$ and $\text{FMap}_{\sigma_1} \neq \text{FMap}_{\sigma_2}$.*

Algorithm VerifyPSO. We are now ready to describe our algorithm VerifyPSO for the problem VPSO-rf. In high level, the algorithm enumerates all lower sets of $(\mathcal{L}(X), \text{PO})$, i.e., the lower sets of the thread events. The crux of the algorithm is to guarantee that for every witness-prefix σ', the algorithm constructs a trace σ such that (i) $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, (ii) $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and (iii) $\text{FMap}_\sigma \leq \text{FMap}_{\sigma'}$. To achieve this, for a given lower set Y of $(\mathcal{L}(X), \text{PO})$, the algorithm examines at most as many traces σ with $\mathcal{L}(\sigma) = Y$ as the number of different fence maps of witness prefixes with the same set of thread events. Hence, the algorithm examines significantly fewer traces than the $n^{k \cdot (d+1)}$ lower sets of (X, PO).

Algorithm 2 presents a formal description of VerifyPSO. The algorithm maintains a worklist $\mathcal{S}$ of prefixes, and a set Done of explored pairs "(thread events, fence map)". Consider an iteration of the main loop in Line 2. First in the loop of Line 4 all spurious executable memory-writes are executed. Then Line 6 checks whether the witness is complete. In case it is not complete, the loop in Line 7 enumerates the possibilities to extend with a thread event. Crucially, the condition in Line 16 ensures that there are no duplicates with the same pair "(thread events, fence map)".

Algorithm 2: VerifyPSO

Input: An event set X and a reads-from function $\mathrm{RF}\colon \mathcal{R}(X) \to \mathcal{W}(X)$
Output: A witness σ that realizes (X, RF) if (X, RF) is realizable under PSO, else $\sigma = \bot$

$\mathcal{S} \leftarrow \{\epsilon\}$; Done $\leftarrow \{\emptyset\}$
while $\mathcal{S} \neq \emptyset$ **do**
  Extract a trace σ from $\mathcal{S}$
  while $\exists$ *spurious* wM PSO-*executable in* σ **do**
    $\sigma \leftarrow \sigma \circ \mathrm{wM}$ // Flush spurious memory-write wM
  if $\mathcal{E}(\sigma) = X$ **then return** σ // Witness found
  foreach *thread event* e PSO-*executable in* σ **do**
    Let $\sigma_e \leftarrow \sigma$
    if e *is a read event with* $\mathrm{RF}(\mathrm{r}) = (\mathrm{wB}, \mathrm{wM})$ **then**
      if $\mathrm{thr}(\mathrm{r}) \neq \mathrm{thr}(\mathrm{wB})$ *and* $\mathrm{wM} \notin \sigma_e$ **then**
        $\sigma_e \leftarrow \sigma_e \circ \mathrm{wM}$ // Execute the reads-from of e
    else if e *is a fence event* **then**
      Let $\mu \leftarrow$ any linearization of $(\mathcal{PW}^M(\sigma, \mathrm{thr}(e)), \mathrm{PO})$
      $\sigma_e \leftarrow \sigma_e \circ \mu$ // Execute pending memory writes
    $\sigma_e \leftarrow \sigma_e \circ e$ // Finally, execute e
    if $\nexists \sigma' \in$ Done *s.t.* $\mathcal{L}(\sigma_e) = \mathcal{L}(\sigma')$ *and* $\mathrm{FMap}_{\sigma_e} = \mathrm{FMap}_{\sigma'}$ **then**
      Insert σ_e in $\mathcal{S}$ and in Done // Continue from σ_e
return $\bot$

Soundness. The soundness of VerifyPSO follows directly from the definition of PSO-executable events, and is similar to the case of VerifyTSO.

Completeness. For each witness prefix σ', algorithm VerifyPSO generates a trace σ with (i) $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, (ii) $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and (iii) $\mathrm{FMap}_\sigma \leq \mathrm{FMap}_{\sigma'}$. This fact directly implies completeness, and it is achieved by the following key invariant. Consider that the algorithm has constructed a trace σ, and is attempting to extend σ with a thread event e. Further, let σ' be an arbitrary witness prefix with (i) $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, (ii) $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and (iii) $\mathrm{FMap}_\sigma \leq \mathrm{FMap}_{\sigma'}$. If σ' can be extended so that the next thread event is e, then e is also executable in σ, and (by Lemma 4.2 and Lemma 4.3) the extension of σ with e maintains the invariant. In Figure 6 we provide an intuitive illustration of the completeness idea.

We now prove the argument in detail for the above σ, σ' and thread event e. Assume that $\sigma' \circ \kappa \circ e$ is a witness prefix as well, for a sequence of memory-writes κ. Consider the following cases.

(1) If e is a read event, let $\mathbf{w} = (\mathrm{wB}, \mathrm{wM}) = \mathrm{RF}(e)$. If it is a local write (i.e., $\mathrm{thr}(\mathbf{w}) = \mathrm{thr}(e)$), necessarily $\mathrm{wB} \in \sigma' \circ \kappa$, and since the traces agree on thread events, we have $\mathrm{wB} \in \sigma$; thus e is executable in σ. Otherwise, $\mathbf{w}$ is a remote write (i.e., $\mathrm{thr}(\mathbf{w}) \neq \mathrm{thr}(e)$). Assume towards contradiction that e is not executable in σ; this can happen in two cases.
In the first case, the variable $x = \mathrm{var}(e)$ is held by another (non-spurious) memory-write wM' in σ. Since $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, the variable x is also held by wM' in $\sigma' \circ \kappa$. But then, both wM and wM' hold x in $\sigma' \circ \kappa$, a contradiction.
In the second case, there is a write $(\mathrm{wB}', \mathrm{wM}')$ with $\mathrm{var}(\mathrm{wM}') = \mathrm{var}(e)$ and $\mathrm{wB}' <_{\mathrm{PO}} e$ and $\mathrm{wM}' \notin \sigma$. If $\mathrm{wM}' \notin \sigma' \circ \kappa$, then e would read-from wB' from the buffer in $\sigma' \circ \kappa \circ e$, contradicting $\mathrm{RF}(e) = (\_, \mathrm{wM})$. Thus $\mathrm{wM}' \in \sigma' \circ \kappa$, and further $\mathrm{wM} \in \sigma' \circ \kappa$ with $\mathrm{wM}' <_{\sigma' \circ \kappa} \mathrm{wM}$. Since $\sigma' \circ \kappa \circ e$ is a witness prefix and $\mathrm{wB}' <_{\mathrm{PO}} e$, we have $\mathrm{wB}' \in \sigma'$. From this and $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$ we have that $\mathrm{wB}' \in \sigma$ and wM' is pending in σ. This together gives us that wM' is spurious in σ. Consider the earliest memory-write pending in σ on the same buffer (i.e., $\mathrm{thr}(\mathrm{wM}')$ and

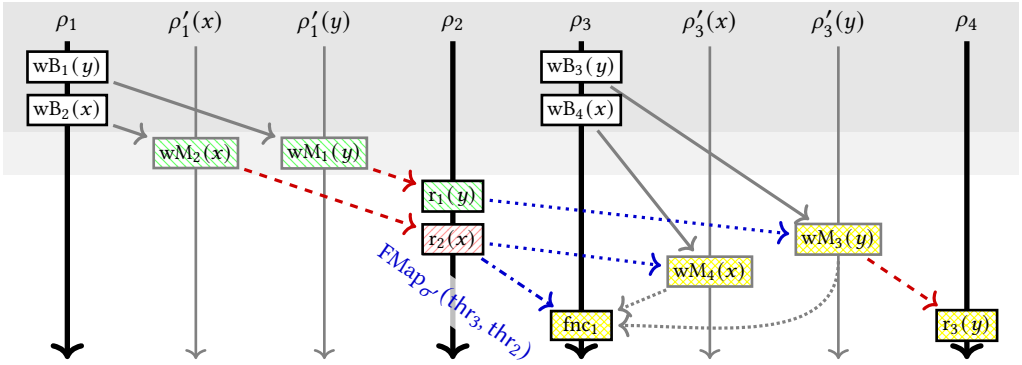


Fig. 6. VerifyPSO completeness idea. Consider the witness prefix σ' (lighter gray) and the corresponding trace σ constructed by the algorithm (darker gray). The fence fnc_1 is PSO-executable in σ but not in σ', since in the latter, $\mathrm{thr}(\mathrm{fnc}_1)$ has non-empty buffers, but the variables x and y are held by wM_1 and wM_2, respectively. This is equivalent to waiting until after r_1 and r_2 have been executed. Since executing r_2 implies having executed r_1, the fence map $\mathrm{FMap}_{\sigma'}(\mathrm{thr}_3, \mathrm{thr}_2)$ compresses this information by only pointing to r_2.

$\mathrm{var}(\mathrm{wM}')$), denote it wM''. We have that $\mathrm{wM}'' \leq_{\mathrm{PO}} \mathrm{wM}'$ and wM'' is spurious in σ. Further, wM'' is executable in σ. But then it would have been added to σ in the while loop of Line 4, a contradiction.

(2) Assume that e is a fence event, and let $\mathrm{wM}_1, \ldots, \mathrm{wM}_j$ be the pending memory-writes of $\mathrm{thr}(e)$ in σ. Suppose towards contradiction that e is not executable. Then one of the wM_i is not executable, let $x = \mathrm{var}(\mathrm{wM}_i)$. Similarly to the above, there can be two cases where this might happen.
The first case is when wM_i must be read-from by some read event $\mathrm{r} \notin \sigma$, but r is preceded by a local write $(\mathrm{wB}, \mathrm{wM})$ (i.e., $\mathrm{wB} <_{\mathrm{PO}} \mathrm{r}$) on the same variable x while $\mathrm{wM} \notin \sigma$. A similar analysis to the previous case shows that the earliest pending write on $\mathrm{thr}(\mathrm{wM})$ for variable x is spurious, and thus already added to σ due to the while loop in Line 4, a contradiction.
The second case is when the variable x is held in σ. Since $\mathrm{FMap}_{\sigma} \leq \mathrm{FMap}_{\sigma'}$, the variable x is also held in σ', and thus wM_i is not executable in σ' either. But then $\sigma' \circ \kappa \circ e$ cannot be a witness prefix, a contradiction.

The following lemma states the correctness of VerifyPSO, which together with the complexity argument establishes Theorem 3.2.

Lemma 4.5. (X, RF) *is realizable under* PSO *iff* VerifyPSO *returns a trace* $\sigma \neq \epsilon$.

We conclude this section with some insights on the relationship between VTSO-rf and VPSO-rf.

Relation between TSO and PSO verification. In high level, TSO might be perceived as a special case of PSO, where every thread is equipped with one buffer (TSO) as opposed to one buffer per global variable (PSO). However, the communication patterns between TSO and PSO are drastically different. As a result, our algorithm VerifyPSO is not applicable to TSO, and we do not see an extension of VerifyTSO for handling PSO efficiently. In particular, the minimal strategy of VerifyPSO on memory-writes is based on the following observation: for a read r observing a remote memory-write wM, it always suffices to execute wM exactly before executing r (unless wM has already been executed). This holds because the corresponding buffer contains memory-writes *only* on the same variable, and thus all such memory-writes that precede wM cannot be read-from by any subsequent read. This property does not hold for TSO: as there is a single buffer, wM might be executed as a result of flushing the buffer of thread $\mathrm{thr}(\mathrm{wM})$ to make another memory-write wM' visible, on a *different* variable than $\mathrm{var}(\mathrm{wM})$, and thus wM' might be observable by a subsequent read. Hence the minimal strategy of VerifyPSO on memory-writes does not apply to TSO. On the

other hand, the maximal strategy of VerifyTSO is not effective for PSO, as it requires enumerating all lower sets of $(\mathcal{W}^M(X), \mathsf{RF})$, which are $n^{k \cdot d}$ many in PSO (where d is the number of variables), and thus this leads to worse bounds than the ones we achieve in Theorem 3.2.

4.3 Closure for VerifyTSO and VerifyPSO

In this section we introduce *closure*, a practical heuristic to efficiently detect whether a given instance (X, RF) of the verification problem VTSO-rf resp. VPSO-rf is unrealizable. Closure is sound, meaning that a realizable instance (X, RF) is never declared unrealizable by closure. Further, closure is not complete, which means there exist unrealizable instances (X, RF) not detected as such by closure. Finally, closure can be computed in time polynomial with respect to the number of events (i.e., size of X), irrespective of the underlying number of threads and variables.

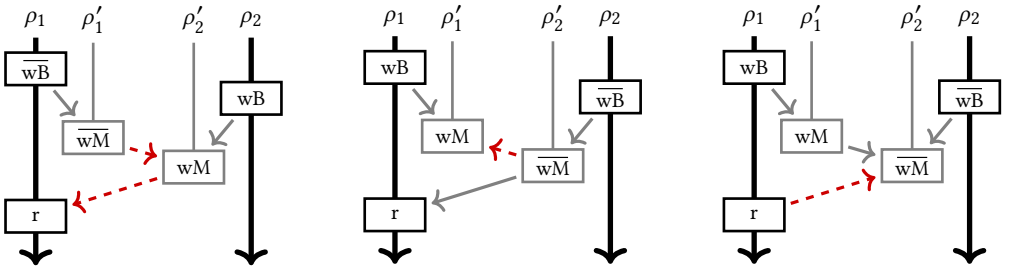
Given an instance (X, RF), any solution of VTSO-rf/VPSO-rf(X, RF) respects $\mathsf{PO}|X$, i.e., the program order upon X. Closure constructs the weakest partial order $P(X)$ that refines the program order (i.e., $P \sqsubseteq \mathsf{PO}|X$) and further satisfies for each read $\mathrm{r} \in \mathcal{R}(X)$ with $\mathsf{RF}(\mathrm{r}) = (\mathrm{wB}, \mathrm{wM})$:

(1) If $\mathsf{thr}(\mathrm{r}) \neq \mathsf{thr}(\mathsf{RF}(\mathrm{r}))$, then (i) $\mathrm{wM} <_P \mathrm{r}$ and (ii) $\overline{\mathrm{wM}} <_P \mathrm{wM}$ for any $(\overline{\mathrm{wB}}, \overline{\mathrm{wM}}) \in \mathcal{W}(X_{\mathsf{thr}(\mathrm{r})})$ such that $\overline{\mathrm{wM}} \bowtie \mathrm{r}$ and $\overline{\mathrm{wB}} <_{\mathsf{PO}} \mathrm{r}$.

(2) For any $\overline{\mathrm{wM}} \in \mathcal{W}^M(X_{\neq \mathsf{thr}(\mathrm{r})})$ such that $\overline{\mathrm{wM}} \bowtie \mathrm{r}$ and $\overline{\mathrm{wM}} \neq \mathrm{wM}$, $\overline{\mathrm{wM}} <_P \mathrm{r}$ implies $\overline{\mathrm{wM}} <_P \mathrm{wM}$.

(3) For any $\overline{\mathrm{wM}} \in \mathcal{W}^M(X_{\neq \mathsf{thr}(\mathrm{r})})$ such that $\overline{\mathrm{wM}} \bowtie \mathrm{r}$ and $\overline{\mathrm{wM}} \neq \mathrm{wM}$, $\mathrm{wM} <_P \overline{\mathrm{wM}}$ implies $\mathrm{r} <_P \overline{\mathrm{wM}}$.

If no above P exists, the instance VTSO-rf/VPSO-rf(X, RF) provably has no solution. In case P exists, each solution σ of VTSO-rf/VPSO-rf(X, RF) provably respects P (formally, $\sigma \sqsubseteq P$).



(a) Rule Item 1. Both new orderings are necessary, as a reversal of either of them would "hide" wM from r, making it impossible for r to read-from (wB, wM).

(b) Rule Item 2. The new ordering is necessary; its reversal would make $\overline{\mathrm{wM}}$ appear between (wB, wM) and r, making it impossible for r to read-from (wB, wM).

(c) Rule Item 3. The new ordering is necessary; its reversal would make $\overline{\mathrm{wM}}$ appear between (wB, wM) and r, making it impossible for r to read-from (wB, wM).

Fig. 7. Illustration of the three closure rules. In each example, the read r has to read-from the write (wB, wM), i.e., $\mathsf{RF}(\mathrm{r}) = (\mathrm{wB}, \mathrm{wM})$. All depicted events are on the same variable (which is omitted for clarity). The gray solid edges illustrate orderings already present in the partial order, and the red dashed edges illustrate the resulting new orderings enforced by the specific rule.

The intuition behind closure is as follows. The construction starts with the program order $\mathsf{PO}|X$, and then, utilizing the above rules Item 1, Item 2 and Item 3, it iteratively adds further event orderings such that every witness execution provably has to follow the orderings. Consequently, if the added orderings induce a cycle, this serves as a proof that there exists no witness of the input instance (X, RF). The rules Item 1, Item 2 and Item 3 can intuitively be though of as simple reasoning arguments why specific orderings have to be present in each witness of (X, RF), and Figure 7 provides an illustration of the rules.

We leverage the guarantees of closure by computing it before executing VerifyTSO resp. VerifyPSO. If no closure P of (X, RF) exists, the algorithm VerifyTSO resp. VerifyPSO does not

need to be executed at all, as we already know that (X, RF) is unrealizable. Otherwise we obtain the closure P, we execute VerifyTSO/VerifyPSO to search for a witness of (X, RF), and we restrict VerifyTSO/VerifyPSO to only consider prefixes σ' respecting P (formally, $\sigma' \sqsubseteq P|\mathcal{E}(\sigma')$), since we know that each solution of VTSO-rf/VPSO-rf(X, RF) has to respect P.

The notion of closure, its beneficial properties, as well as construction algorithms are well-studied for the SC memory model [Abdulla et al. 2019; Chalupa et al. 2017; Pavlogiannis 2019]. Our conditions above extend this notion to TSO and PSO. Moreover, the closure we introduce here is *complete* for concurrent programs with two threads, i.e., if P exists then there is a valid trace realizing (X, RF) under the respective memory model.

4.4 Verifying Executions with Atomic Primitives

For clarity of presentation of the core algorithmic concepts, we have thus far neglected more involved atomic operations, namely atomic *read-modify-write* (RMW) and atomic *compare-and-swap* (CAS). We show how our approach handles verification of TSO and PSO executions that also include RMW and CAS operations here in a separate section. Importantly, our treatment retains the complexity bounds established in Theorem 3.1 and Theorem 3.2.

Atomic instructions. We consider the concurrent program under the TSO resp. PSO memory model, which can further atomically execute the following types of instructions.

(1) A *read-modify-write* instruction rmw executes atomically the following sequence. It (i) reads, with respect to the TSO resp. PSO semantics, the value v of a global variable $x \in \mathcal{G}$, then (ii) uses v to compute a new value v', and finally (iii) writes the new value v' to the global variable x. An example of a typical rmw computation is fetch-and-add (resp. fetch-and-sub), where $v' = v + c$ for some positive (resp. negative) constant c.
(2) A *compare-and-swap* instruction cas executes atomically the following sequence. It (i) reads, with respect to the TSO resp. PSO semantics, the value v of a global variable $x \in \mathcal{G}$, (ii) compares it with a value c, and (iii) if $v = c$ then it writes a new value v' to the global variable x.

Each instruction of the above two types blocks (i.e., it cannot get executed) until the buffer of its thread is empty (resp. all buffers of its thread are empty in PSO). Finally, the instruction specifies the nature of its final write. This write is either enqueued into its respective buffer (to be dequeued into shared memory at a later point), or it gets immediately flushed into the shared memory.

Atomic instructions modeling. In our approach we handle atomic RMW and CAS instructions without introducing them as new event types. Instead, we model these instructions as sequences of already considered events, i.e., reads, buffer-writes, memory-writes, and fences. We annotate some events of an atomic instruction to constitute an *atomic block*, which intuitively indicates that the event sequence of the atomic block cannot be interleaved with other events, thus respecting the semantics of the instruction.

(1) A *read-modify-write* instruction rmw on a variable x is modeled as a sequence of four events: (i) a fence event, (ii) a read of x, (iii) a buffer-write of x, and (iv) a memory-write of x. The read and buffer-write events (ii)+(iii) are annotated as constituting an atomic block; in case the write of rmw is specified to proceed immediately to the shared memory, the memory-write event (iv) is also part of the atomic block.
(2) For a *compare-and-swap* instruction cas we consider separately the following two cases. A *successful* cas (i.e., the write proceeds) is modeled the same way as a read-modify-write. A *failed* cas (i.e., the write does not proceed) is modeled simply as a fence followed by a read, with no atomic block.

Executable atomic blocks. Here we describe the TSO- and PSO-executability conditions for an atomic block. No further additions for executability are required, since no new event types are introduced to handle RMW and CAS instructions.

Consider an instance (X, RF) of VTSO-rf, and a trace σ with $\mathcal{E}(\sigma) \subseteq X$. An atomic block containing a sequence of events $e_1, ..., e_j$ is TSO-executable in σ if:

(1) for each $1 \leq i \leq j$ we have that $e_i \in X \setminus \mathcal{E}(\sigma)$, and
(2) for each $1 \leq i \leq j$ we have that e_i is TSO-executable in $\sigma \circ e_1 ... e_{i-1}$.

Intuitively, an atomic block is TSO-executable if it can be executed as a sequence at once (i.e., without other events interleaved), and the TSO-executable conditions of each event (i.e., a read or a buffer-write or a memory-write or a fence) within the block are respected.

The PSO-executable conditions are analogous. Given an instance (X, RF) of VPSO-rf and a trace σ with $\mathcal{E}(\sigma) \subseteq X$, an atomic block of events $e_1, ..., e_j$ is PSO-executable in σ if:

(1) for each $1 \leq i \leq j$ we have that $e_i \in X \setminus \mathcal{E}(\sigma)$, and
(2) for each $1 \leq i \leq j$ we have that e_i is PSO-executable in $\sigma \circ e_1 ... e_{i-1}$.

Execution verification. Given the above executable conditions, the execution verification algorithms VerifyTSO and VerifyPSO only require minor technical modifications to verify executions including RMW and CAS instructions.

The core idea of the VerifyTSO resp. VerifyPSO modifications is to not extend prefixes with single events that are part of some atomic block, and instead extend the atomic blocks fully. This way, a lower set of (X, PO) is considered only if for each atomic block, the block is either fully present or fully not present in the lower set.

In VerifyTSO (Algorithm 1), in Line 4 we further consider each TSO-executable atomic block $e_1, ..., e_j$ not containing any memory-write event, and then in Line 5 we extend the prefix with the entire atomic block, i.e., $\sigma \leftarrow \sigma \circ e_1, ..., e_j$. Further, in Line 7 we further consider each TSO-executable atomic block $e_1, ..., e_j$ containing a memory-write event, and in Line 8 we then extend the prefix with the whole atomic block, i.e., $\sigma \leftarrow \sigma \circ e_1, ..., e_j$.

In VerifyPSO (Algorithm 2), in the loop of Line 7 we further consider each PSO-executable atomic block. Consider a fixed iteration of this loop with an atomic block $e_1, ..., e_j$. The first event of the atomic block e_1 is a read, thus the condition in Line 9 is evaluated true with e_1 and the control flow moves to Line 10. Later, the condition in Line 12 is evaluated false (since e_1 is a read). Finally, in Line 15 the prefix is extended with the whole atomic block, i.e., $\sigma_e \leftarrow \sigma_e \circ e_1, ..., e_j$.

For VerifyTSO the argument of maintaining maximality in the set of thread events applies also in the presence of RMW and CAS, and thus the bound of Theorem 3.1 is retained. Similarly, for VerifyPSO the enumeration of fence maps and the maximality in the spurious writes is preserved also with RMW and CAS, and hence the bound of Theorem 3.2 holds.

Closure. When verifying executions with RMW and CAS instructions, while the closure retains its guarantees as is, it can more effectively detect unrealizable instances with additional rules. Specifically, the closure P of (X, RF) satisfies the rules 1–3 described in Section 4.3, and additionally, given an event e and an atomic block $e_1, ..., e_j$, P satisfies the following.

(4) If $e_i <_P e$ for any $1 \leq i \leq j$, then $e_j <_P e$ (i.e., if some part of the block is before e then the entire block is before e).
(5) If $e <_P e_i$ for any $1 \leq i \leq j$, then $e <_P e_1$ (i.e., if e is before some part of the block then e is before the entire block).

5 READS-FROM SMC FOR TSO AND PSO

In this section we present RF-SMC, an exploration-optimal reads-from SMC algorithm for TSO and PSO. The algorithm RF-SMC is based on the reads-from algorithm for SC [Abdulla et al. 2019], and

adapted in this work to handle the relaxed memory models TSO and PSO. The algorithm uses as subroutines VerifyTSO (resp. VerifyPSO) to decide whether any given class of the RF partitioning is consistent under the TSO (resp. PSO) semantics.

RF-SMC is a recursive algorithm, each call of RF-SMC is argumented by a tuple $(\tau, \mathsf{RF}, \sigma, \mathsf{mrk})$ where the following points hold:

- τ is a sequence of thread events. Let X denote the set of events of τ together with their memory-write counterparts, formally $X = \mathcal{E}(\tau) \cup \{\mathrm{wM} : \exists(\mathrm{wB}, \mathrm{wM}) \in \mathcal{W} \text{ such that } \mathrm{wB} \in \mathcal{W}^B(\tau)\}$.
- $\mathsf{RF} : \mathcal{R}(X) \to \mathcal{W}(X)$ is a desired reads-from function.
- σ is a concrete valid trace that is a witness of (X, RF), i.e., $\mathcal{E}(\sigma) = X$ and $\mathsf{RF}_\sigma = \mathsf{RF}$.
- $\mathsf{mrk} \subseteq \mathcal{R}(\tau)$ is a set of reads that are *marked* to be committed to the source they read-from in σ.

Further, a globally accessible set of schedule sets called schedules is maintained throughout the recursion. The schedules set is initialized empty ($\mathsf{schedules} = \emptyset$) and the initial call of the algorithm is argumented with empty sequences and sets — $\text{RF-SMC}(\epsilon, \emptyset, \epsilon, \emptyset)$.

Algorithm 3: $\text{RF-SMC}(\tau, \mathsf{RF}, \sigma, \mathsf{mrk})$

Input: Sequence τ, desired reads-from RF, valid trace σ such that $\mathsf{RF}_\sigma = \mathsf{RF}$, marked reads mrk.

1 $\widetilde{\sigma} \leftarrow \sigma \circ \widehat{\sigma}$ where $\widehat{\sigma}$ is an arbitrary maximal extension of σ // Maximally extend trace σ
2 $\widetilde{\tau} \leftarrow \tau \circ \widehat{\sigma}|\mathcal{L}(\widehat{\sigma})$ // Extend τ with the thread-events subsequence of the extension $\widehat{\sigma}$
3 **foreach** $\mathrm{r} \in \mathcal{R}(\widehat{\sigma})$ **do** // Reads of the extension $\widehat{\sigma}$
4 $\quad$ $\mathsf{schedules}(\mathrm{pre}_{\widetilde{\tau}}(\mathrm{r})) \leftarrow \emptyset$ // Initialize new schedule set
5 **foreach** $\mathrm{r} \in \mathcal{R}(\widetilde{\tau}) \setminus \mathsf{mrk}$ **do** // Unmarked reads
6 $\quad$ $P \leftarrow \mathrm{PO}|\mathcal{E}(\widetilde{\sigma})$ // Program order on all the events of $\widetilde{\sigma}$
7 $\quad$ **foreach** $\mathrm{r}' \in \mathcal{R}(\widetilde{\tau}) \setminus \{\mathrm{r}\}$ *with* $\mathrm{thr}(\mathrm{r}') \neq \mathrm{thr}(\mathsf{RF}_{\widetilde{\sigma}}(\mathrm{r}'))$ **do** // Different-thread-$\mathsf{RF}_{\widetilde{\sigma}}$ reads except r
8 $\quad\quad$ insert $\mathrm{wM} \to \mathrm{r}'$ into P where $\mathsf{RF}_{\widetilde{\sigma}}(\mathrm{r}') = \mathrm{wM}$ // Add the reads-from ordering into P
9 $\quad$ $\mathsf{mutations} \leftarrow \{(\mathrm{wB}, \mathrm{wM}) \in \mathcal{W}(\widetilde{\sigma}) \mid \mathrm{r} \bowtie \mathrm{wM}\} \setminus \{\mathsf{RF}_{\widetilde{\sigma}}(\mathrm{r})\}$ // All different writes r may read-from
10 $\quad$ **if** $\mathrm{r} \notin \mathcal{R}(\widehat{\sigma})$ **then** // If r is not part of the extension then
11 $\quad\quad$ $\mathsf{mutations} \leftarrow \mathsf{mutations} \cap \mathcal{W}(\widehat{\sigma})$ // Only consider writes of the extension
12 $\quad$ **foreach** $(\mathrm{wB}, \mathrm{wM}) \in \mathsf{mutations}$ **do** // Considered mutations
13 $\quad\quad$ $\mathsf{causesafter} \leftarrow \{e \in \mathcal{E}(\widetilde{\tau}) \mid \mathrm{r} <_{\widetilde{\tau}} e \text{ and } e \leq_P \mathrm{wB}\}$ // Causal past of wB after r in $\widetilde{\tau}$
14 $\quad\quad$ $\tau' \leftarrow \mathrm{pre}_{\widetilde{\tau}}(\mathrm{r}) \circ \widetilde{\tau}|\mathsf{causesafter}$ // r-prefix followed by causesafter
15 $\quad\quad$ $X' \leftarrow \mathcal{E}(\tau') \cup \{\mathrm{wM}' : (\mathrm{wB}', \mathrm{wM}') \in \mathcal{W}(\widetilde{\sigma}) \text{ and } \mathrm{wB}' \in \mathcal{W}^B(\tau')\}$ // Event set for this mutation
16 $\quad\quad$ $\mathsf{RF}' \leftarrow \{(\mathrm{r}', \mathsf{RF}_{\widetilde{\sigma}}(\mathrm{r}')) : \mathrm{r}' \in \mathcal{R}(\tau') \text{ and } \mathrm{r}' \neq \mathrm{r}\} \cup \{(\mathrm{r}, (\mathrm{wB}, \mathrm{wM}))\}$ // Reads-from for this mutation
17 $\quad\quad$ **if** $(\tau', \mathsf{RF}', \_, \_) \notin \mathsf{schedules}(\mathrm{pre}_{\widetilde{\tau}}(\mathrm{r}))$ **then** // If this is a new schedule
18 $\quad\quad\quad$ $\sigma' \leftarrow \mathrm{Witness}(X', \mathsf{RF}')$ // VerifyTSO (Algorithm 1) or VerifyPSO (Algorithm 2)
19 $\quad\quad\quad$ **if** $\sigma' \neq \perp$ **then** // If the mutation is realizable
20 $\quad\quad\quad\quad$ $\mathsf{mrk}' \leftarrow (\mathsf{mrk} \cap \mathcal{R}(\tau')) \cup \mathcal{R}(\mathsf{causesafter})$ // Reads in causesafter get newly marked
21 $\quad\quad\quad\quad$ add $(\tau', \mathsf{RF}', \sigma', \mathsf{mrk}')$ to $\mathsf{schedules}(\mathrm{pre}_{\widetilde{\tau}}(\mathrm{r}))$ // Add the successful new schedule
22 **foreach** $\widehat{\mathrm{r}} \in \mathcal{R}(\widehat{\sigma})$ *in the reverse order of* $<_{\widehat{\sigma}}$ **do** // Extension reads starting from the end
23 $\quad$ **foreach** $(\tau', \mathsf{RF}', \sigma', \mathsf{mrk}') \in \mathsf{schedules}(\mathrm{pre}_{\widetilde{\tau}}(\widehat{\mathrm{r}}))$ **do** // Collected schedules mutating $\widehat{\mathrm{r}}$
24 $\quad\quad$ $\text{RF-SMC}(\tau', \mathsf{RF}', \sigma', \mathsf{mrk}')$ // Recursive call on the schedule
25 $\quad$ delete $\mathsf{schedules}(\mathrm{pre}_{\widetilde{\tau}}(\widehat{\mathrm{r}}))$ // This schedule set has been fully explored, hence it can be deleted

Algorithm 3 presents the pseudocode of RF-SMC. In each call of RF-SMC, a number of possible changes (or *mutations*) of the desired reads-from function RF is proposed in iterations of the loop in Line 5. Consider the read r of a fixed iteration of the Line 5 loop. First, in Lines 6–8 a partial order P is constructed to capture the causal past of write events. In Lines 9–11 the set of mutations for r is computed. Then in each iteration of the Line 12 loop a mutation is constructed (Lines 13–16). Here the partial order P is utilized in Line 13 to help determine the event set of the mutation. The constructed mutation, if deemed novel (checked in Line 17), is probed whether it is realizable (in

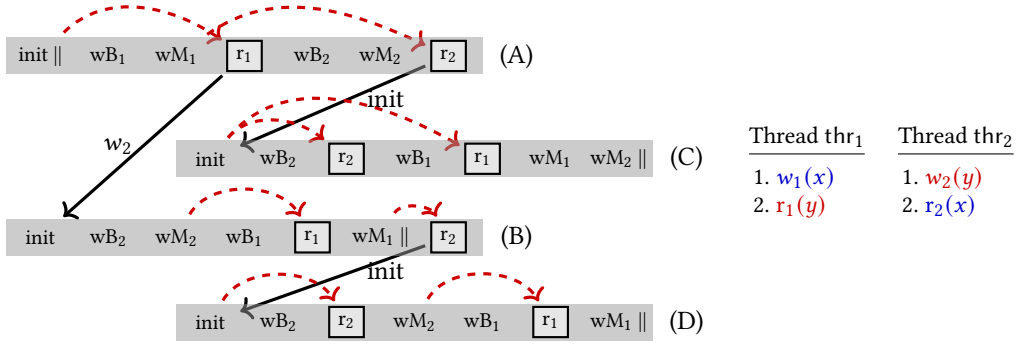


Fig. 8. RF-SMC (Algorithm 3). The gray boxes represent individual calls to RF-SMC. The sequence of events inside a gray box is the trace $\widetilde{\sigma}$; the part left of the ||-separator is σ (before extending), and to the right is $\widehat{\sigma}$ (the extension). The red dashed arrows represent the reads-from function $\mathrm{RF}_{\widetilde{\sigma}}$. Each black solid arrow represents a recursive call, where the arrow's outgoing tail and label describes the corresponding mutation.

Line 18). In case it is realizable, it gets added into schedules in Line 21. After all the mutations are proposed, then in Lines 22–25 a number of recursive calls of RF-SMC is performed, and the recursive RF-SMC calls are argumented by the specific schedules retrieved.

Figure 8 illustrates the run of RF-SMC on a simple concurrent program (the run is identical under both TSO and PSO). An initial trace (A) is obtained where $\mathrm{r}_1(y)$ reads-from the initial event and $\mathrm{r}_2(x)$ reads-from $w_1(x)$. Here two mutations are probed and both are realizable. In the first mutation (B), $\mathrm{r}_1(y)$ is mutated to read-from $w_2(y)$ and $\mathrm{r}_2(x)$ is not retained (since it appears after $\mathrm{r}_1(y)$ and it is not in the causal past of $w_2(y)$). In the second mutation (C), $\mathrm{r}_2(x)$ is mutated to read-from the initial event and $\mathrm{r}_1(y)$ is retained (since it appears before $\mathrm{r}_2(x)$) with initial event as its reads-from. After both mutations are added to schedules, recursive calls are performed in the reverse order of reads appearing in the trace, thus starting with (C). Here no mutations are probed since there are no events in the extension, the algorithm backtracks to (A) and a recursive call to (B) is performed. Here one mutation (D) is added, where $\mathrm{r}_2(x)$ is mutated to read-from the initial event and $\mathrm{r}_1(y)$ is retained (it appears before $\mathrm{r}_2(x)$) with $w_2(y)$ as its reads-from. The call to (D) is performed and here no mutations are probed (there are no events in the extension). The algorithm backtracks and concludes, exploring four RF partitioning classes in total.

RF-SMC is sound, complete and exploration-optimal, and we formally state this in Theorem 3.3.

Extension from SC to TSO and PSO. The fundamental challenge in extending the SC algorithm of Abdulla et al. [2019] to TSO and PSO is verifying execution consistency for TSO and PSO, which we address in Section 4 (Line 18 of Algorithm 3 calls our algorithms VerifyTSO and VerifyPSO). The main remaining challenge is then to ensure that the exploration optimality is preserved. To that end, we have to exclude certain events (in particular, memory-write events) from subsequences and event subsets that guide the exploration of Algorithm 3. Specifically, the sequences τ, τ', and $\widetilde{\tau}$ invariantly contain only the thread events, which is ensured in Line 2, Line 13 and Line 14, and then in Line 15 the absent memory-writes are reintroduced. No such distinction is required under SC.

Remark 1 (Handling locks and atomic primitives). For clarity of presentation, so far we have neglected locks in our model. However, lock events can be naturally handled by our approach as follows. We consider each lock-release event release as an atomic write event (i.e., its effects are not deferred by a buffer but instead are instantly visible to each thread). Then, each lock-acquire event acquire is considered as a read event that accesses the unique memory location.

In SMC, we enumerate the reads-from functions that also consider locks, thus having constraints of the form RF(acquire) = release. This treatment totally orders the critical sections of each lock, which naturally solves all reads-from constraints of locks, and further ensures that no thread acquires an already acquired (and so-far unreleased) lock. Therefore VerifyTSO/VerifyPSO need not take additional care for locks. The approach to handle locks by Abdulla et al. [2019] directly carries over to our exploration algorithm RF-SMC.

The atomic operations read-modify-write (RMW) and compare-and-swap (CAS) are modeled as in Section 4.4, except for the fact that the atomic blocks are not necessary for SMC. Then RF-SMC can handle programs with such operations as described by Abdulla et al. [2019]. In particular, the modification of RF-SMC (Algorithm 3) to handle RMW and CAS operations is as follows.

Consider an iteration of the loop in Line 5 where r is the read-part of either a RMW or a successful CAS, denoted e, and let $(\mathrm{wB}'', \mathrm{wM}'') = \mathrm{RF}_{\widetilde{\sigma}}(\mathrm{r})$. Then, in Line 9 we additionally consider as an extra mutation each atomic instruction e' satisfying:

(1) The read-part r' of e' reads-from the write-part $(\mathrm{wB}, \mathrm{wM})$ of e (i.e., $\mathrm{RF}_{\widetilde{\sigma}}(\mathrm{r}') = (\mathrm{wB}, \mathrm{wM})$), and
(2) e' is either a RMW, or it will be a successful CAS when it reads-from $(\mathrm{wB}'', \mathrm{wM}'')$. In this case, let $(\mathrm{wB}', \mathrm{wM}')$ denote the write-part of e'.

When considering the above mutation in Line 12, we set $\mathrm{RF}'(\mathrm{r}') = (\mathrm{wB}'', \mathrm{wM}'')$ and $\mathrm{RF}'(\mathrm{r}) = (\mathrm{wB}', \mathrm{wM}')$ in Line 16, which intuitively aims to "reverse" e and e' in the trace.

6 EXPERIMENTS

In this section we report on an experimental evaluation of the consistency verification algorithms VerifyTSO and VerifyPSO, as well as the reads-from SMC algorithm RF-SMC. We have implemented our algorithms as an extension in Nidhugg [Abdulla et al. 2015], a state-of-the-art stateless model checker for multithreaded C/C++ programs with pthreads library, operating on LLVM IR.

Benchmarks. For our experimental evaluation of both the consistency verification and SMC, we consider 109 benchmarks coming from four different categories, namely: (i) SV-COMP benchmarks, (ii) benchmarks from related papers and works [Abdulla et al. 2015, 2019; Chatterjee et al. 2019; Huang and Huang 2016], (iii) mutual-exclusion algorithms, and (iv) dynamic-programming benchmarks of Chatterjee et al. [2019]. Although the consistency and SMC algorithms can be extended to support atomic compare-and-swap and read-modify-write primitives (cf. Remark 1), our current implementation does not support these primitives. Therefore, we used all benchmarks without such primitives that we could obtain (e.g., we include every benchmark of the relevant SC reads-from work [Abdulla et al. 2019] except the one benchmark with compare-and-swap). Each benchmark comes with a scaling parameter, called the *unroll* bound, which controls the bound on the number of iterations in all loops of the benchmark (and in some cases it further controls the number of threads).

6.1 Experiments on Execution Verification for TSO and PSO

In this section we perform an experimental evaluation of our execution verification algorithms VerifyTSO and VerifyPSO. For the purpose of comparison, we have also implemented within Nidhugg the naive lower-set enumeration algorithm of Abdulla et al. [2019]; Biswas and Enea [2019], extended to TSO and PSO. Intuitively, this approach enumerates all lower sets of the program order restricted to the input event set, which yields a better complexity bound than enumerating write-coherence orders (even with just one location). The extensions to TSO and PSO are called NaiveVerifyTSO and NaiveVerifyPSO, respectively, and their worst-case complexity is $n^{2 \cdot k}$ and $n^{k \cdot (d+1)}$, respectively (as discussed in Section 3). Further, for each of the above verification algorithms, we consider two variants, namely, with and without the closure heuristic of Section 4.3.

Setup. We evaluate the verification algorithms on execution consistency instances induced during SMC of the benchmarks. For TSO we have collected 9400 instances, 1600 of which are not realizable. For PSO we have collected 9250 instances, 1400 of which are not realizable. The collection process is described in detail in Appendix C.1 of Bui et al. [2021]. For each instance, we run the verification algorithms subject to a timeout of one minute, and we report the average time achieved over 5 runs.

Below we present the results using logarithmically scaled plots, where the opaque and semi-transparent red lines represent identity and an order-of-magnitude difference, respectively.

Results – algorithms with closure. Here we evaluate the verification algorithms that execute the closure as the preceding step. The plots in Figure 9 present the results for TSO and PSO.

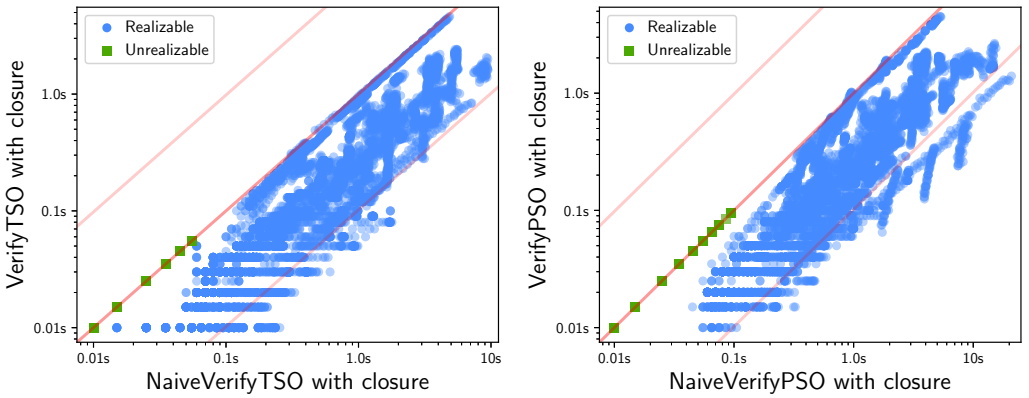


Fig. 9. Consistency verification comparison on TSO (left) and PSO (right) when using closure.

In TSO, our algorithm VerifyTSO is similar to or faster than NaiveVerifyTSO on the realizable instances (blue dots), and the improvement is mostly within an order of magnitude. All unrealizable instances (green dots) were detected as such by closure, and hence the closure-using VerifyTSO and NaiveVerifyTSO coincide on these instances.

We make similar observations in PSO, where VerifyPSO is similar or superior to NaiveVerifyPSO for the realizable instances, and the algorithms are indentical on the unrealizable instances, since these are all detected as unrealizable by closure.

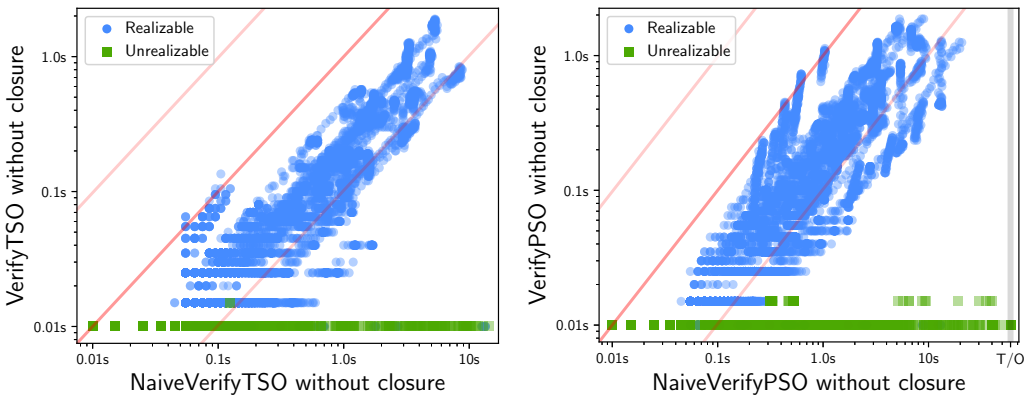


Fig. 10. Consistency verification comparison on TSO (left) and PSO (right) without the closure.

Results – algorithms without closure. Here we evaluate the verification algorithms without the closure. The plots in Figure 10 present the results for TSO and PSO.

In TSO, the algorithm VerifyTSO outperforms NaiveVerifyTSO on most of the realizable instances (blue dots). Further, VerifyTSO significantly outperforms NaiveVerifyTSO on the unrealizable instances (green dots). This is because without closure, a verification algorithm can declare an instance unrealizable only after an exhaustive exploration of its respective lower-set space. VerifyTSO explores a significantly smaller space compared to NaiveVerifyTSO, as outlined in Section 3.

Similar observations as above hold in PSO for the algorithms VerifyPSO and NaiveVerifyPSO without closure, both for the realizable and the unrealizable instances.

Results – effect of closure. Here we comment on the effect of closure for the verification algorithms, in Appendix C.2 of Bui et al. [2021] we present the detailed analysis. Recall that closure constructs a partial order that each witness has to satisfy, and declares an instance unrealizable when it detects that the partial order cannot be constructed for this instance (we refer to Section 4.3 for details).

For each verification algorithm, its version without closure is faster on most instances that are realizable (i.e., a witness exists). This means that the overhead of computing the closure typically outweighs the consecutive benefit of the verification being guided by the partial order.

On the other hand, for each verification algorithm, its version with closure is significantly faster on the unrealizable instances (i.e., no witness exists). This is because a verification algorithm has to enumerate all its lower sets before declaring an instance unrealizable, and this is much slower than the polynomial closure computation.

Results – verification with atomic operations. Here we present additional experiments to evaluate TSO verification algorithms VerifyTSO and NaiveVerifyTSO on executions containing atomic operations read-modify-write (RMW) and compare-and-swap (CAS). To that end, we consider 1088 verification instances (779 realizable and 309 not realizable) that arise during stateless model checking of benchmarks containing RMW and CAS, namely:

- synthetic benchmarks casrot [Abdulla et al. 2019] and cinc [Kokologiannakis et al. 2019b],
- data structure benchmarks barrier, chase-lev, ms-queue and linuxrwlocks [Kokologiannakis et al. 2019b; Norris and Demsky 2013], and
- Linux kernel benchmarks mcs_spinlock and qspinlock [Kokologiannakis et al. 2019b].

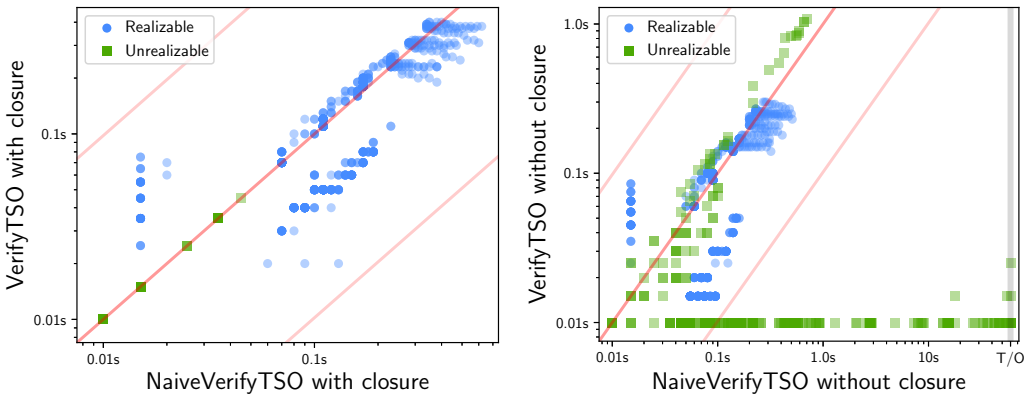


Fig. 11. Consistency verification comparison of VerifyTSO and NaiveVerifyTSO with closure (left) and without closure (right) on verification instances that contain RMW and CAS instructions.

The results are presented in Figure 11. The left plot depicts the results for VerifyTSO and NaiveVerifyTSO when closure is used as a preceding step. Here the results are all within an order-of-magnitude difference, and they are identical for unrealizable instances, since all of them were detected as unrealizable already by the closure. The right plot depicts the results for VerifyTSO and NaiveVerifyTSO without using the closure. Here the difference for realizable instances is also within an order of magnitude, but for some unrealizable instances the algorithm VerifyTSO is significantly faster. Generally, the observed improvement of our VerifyTSO as compared to NaiveVerifyTSO is somewhat smaller in Figure 11, which could be due to the fact that executions with RMW and CAS instructions typically have fewer concurrent writes (indeed, in an execution where each write event is a part of a RMW/CAS instruction, each conflicting pair of writes is inherently ordered by the reads-from orderings together with PO). Finally, in Appendix C.2 of Bui et al. [2021] the effect of closure is evaluated for both verification algorithms VerifyTSO and NaiveVerifyTSO on instances with RMW and CAS.

6.2 Experiments on SMC for TSO and PSO

In this section we focus on assessing the advantages of utilizing the reads-from equivalence for SMC in TSO and PSO. We have used RF-SMC for stateless model checking of 109 benchmarks under each memory model $\mathcal{M} \in \{\text{SC}, \text{TSO}, \text{PSO}\}$, where SC is handled in our implementation as TSO with a fence inserted after each thread event. Appendix C.3 of Bui et al. [2021] provides further details on our SMC setup.

Comparison. As a baseline for comparison, we have also executed Source-DPOR [Abdulla et al. 2014], which is implemented in Nidhugg and explores the trace space using the partitioning based on the Shasha–Snir equivalence. In SC, we have further executed rfsc, the Nidhugg implementation of the reads-from SMC algorithm for SC by Abdulla et al. [2019], and the full comparison that includes rfsc for SC is in Appendix C.4 of Bui et al. [2021]. Both rfsc and Source are well-optimized, and recently started using advanced data-structures for SMC [Lång and Sagonas 2020]. The works of Kokologiannakis et al. [2019b]; Kokologiannakis and Vafeiadis [2020] provide a general interface for reads-from SMC in relaxed memory models. However, they handle a given memory model assuming that an auxiliary consistency verification algorithm for that memory model is provided. No such consistency algorithm for TSO or PSO is presented by Kokologiannakis et al. [2019b]; Kokologiannakis and Vafeiadis [2020], and, to our knowledge, the tool implementations of Kokologiannakis et al. [2019b]; Kokologiannakis and Vafeiadis [2020] also lack a consistency algorithm for both TSO and PSO. Thus these tools are not included in the evaluation.[1]

Evaluation objective. Our objective for the SMC evaluation is three-fold. First, we want to quantify how each memory model $\mathcal{M} \in \{\text{SC}, \text{TSO}, \text{PSO}\}$ impacts the size of the RF partitioning. Second, we are interested to see whether, as compared to the baseline Shasha–Snir equivalence, the RF equivalence leads to coarser partitionings for TSO and PSO, as it does for SC [Abdulla et al. 2019]. Finally, we want to determine whether a coarser RF partitioning leads to faster exploration. Theorem 3.3 states that RF-SMC spends polynomial time per partitioning class, and we aim to see whether this is a small polynomial in practice.

Results. We illustrate the obtained results with several scatter plots. Each plot compares two algorithms executing under specified memory models. Then for each benchmark, we consider the highest attempted unroll bound where both the compared algorithms finish before the one-hour timeout. Green dots indicate that a trace reduction was achieved on the underlying benchmark by the algorithm on the y-axis as compared to the algorithm on the x-axis. Benchmarks with no

[1] Another related work is MCR [Huang and Huang 2016], however, the corresponding tool operates on Java programs and uses heavyweight SMT solvers that require fine tuning, and thus is beyond the experimental scope of this work.

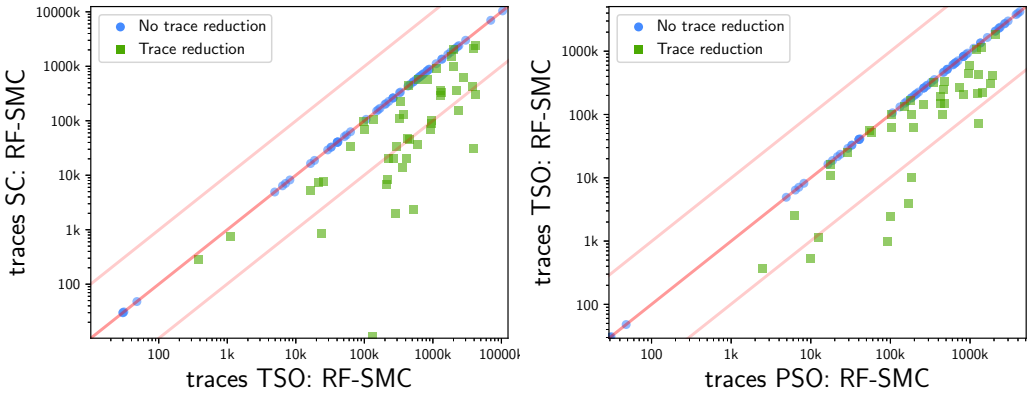


Fig. 12. Traces comparison as RF-SMC moves from SC to TSO (left) and from TSO to PSO (right).

trace reduction are represented by the blue dots. All scatter plots are in log scale, the opaque and semi-transparent red lines represent identity and an order-of-magnitude difference, respectively.

The plots in Figure 12 illustrate how the size of the RF partitioning explored by RF-SMC changes as we move to more relaxed memory models (SC to TSO to PSO). The plots in Figure 13 capture how the size of the RF partitioning explored by RF-SMC relates to the size of the Shasha–Snir partitioning explored by Source. Finally, the plots in Figure 14 demonstrate the time comparison of RF-SMC and Source when there is some (green dots) or no (blue dots) RF-induced trace reduction.

Below we discuss the observations on the obtained results. Table 1 captures detailed results on several benchmarks that we refer to as examples in the discussion.

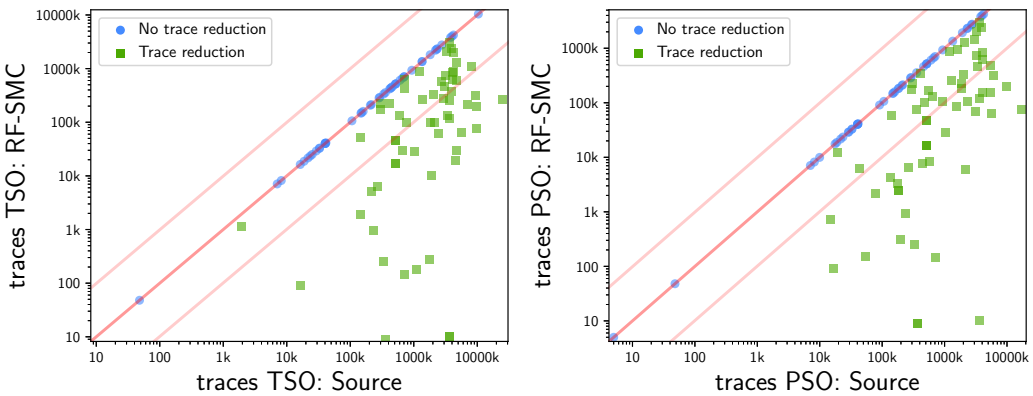


Fig. 13. Traces comparison for RF-SMC and Source on the TSO (left) and PSO (right) memory model.

Discussion. We notice that the analysed programs can often exhibit additional behavior in relaxed memory settings. This causes an increase in the size of the partitionings explored by SMC algorithms (see 27_Boop4 in Table 1 as an example). Figure 12 illustrates the overall phenomenon for RF-SMC, where the increase of the RF partitioning size (and hence the number of traces explored) is sometimes beyond an order of magnitude when moving from SC to TSO, or from TSO to PSO.

We observe that across all memory models, the reads-from equivalence can offer significant reduction in the trace partitioning as compared to Shasha–Snir equivalence. This leads to fewer traces that need to be explored, see the plots of Figure 13. As we move towards more relaxed memory (SC to TSO to PSO), the reduction of RF partitioning often becomes more prominent

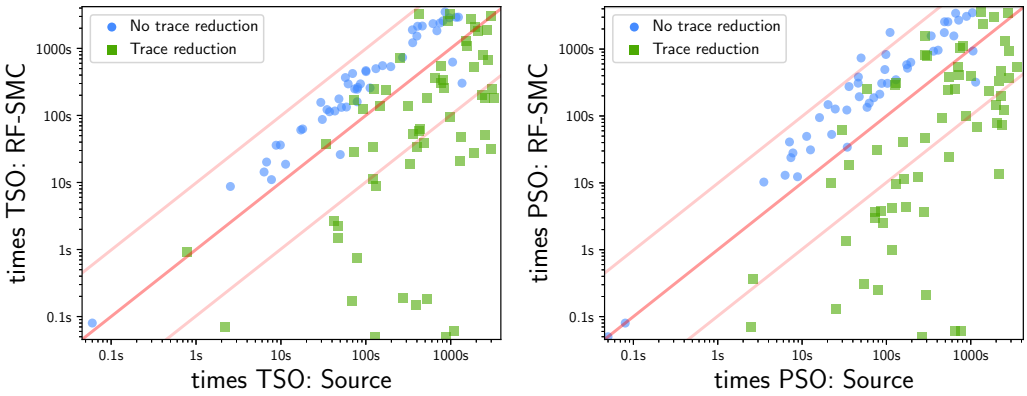


Fig. 14. Times comparison for RF-SMC and Source on the TSO (left) and PSO (right) memory model.

Table 1. SMC results on several benchmarks. **U** denotes the unroll bound. The timeout of one hour is indicated by "-". Bold-font entries indicate the smallest numbers for the respective memory model.

Benchmark		U	Seq. Consistency		Total Store Order		Partial Store Order	
			RF-SMC	Source	RF-SMC	Source	RF-SMC	Source
27_Boop4 threads: 4	Traces	1	**2902**	21948	**3682**	36588	**8233**	572436
		4	**197260**	3873348	**313336**	9412428	**1807408**	-
	Times	1	**1.22s**	1.74s	**1.46s**	6.18s	**4.40s**	169s
		4	**124s**	550s	**182s**	2556s	**1593s**	-
eratosthenes threads: 2	Traces	17	**4667**	100664	**29217**	4719488	**253125**	-
		21	**19991**	1527736	**223929**	-	-	-
	Times	17	**6.70s**	46s	**32s**	2978s	**475s**	-
		21	**41s**	736s	**342s**	-	-	-
fillarray_false threads: 2	Traces	3	**14625**	47892	**14625**	59404	**14625**	63088
		4	**471821**	2278732	**471821**	3023380	**471821**	3329934
	Times	3	12s	**6.18s**	**12s**	**12s**	**18s**	39s
		4	553s	**331s**	**547s**	778s	**930s**	2844s

(see 27_Boop4 in Table 1). Interestingly, in some cases the size of the Shasha–Snir partitioning explored by Source increases as we move to more relaxed settings, while the RF partitioning remains unchanged (cf. `fillarray_false` in Table 1). All these observations signify advantages of RF for analysis of the more complex program behavior that arises due to relaxed memory.

We now discuss how trace partitioning coarseness affects execution time, observing the plots of Figure 14. We see that in cases where RF partitioning is coarser (green dots), our RF algorithm RF-SMC often becomes significantly faster than the Shasha–Snir-based Source, allowing us to analyse programs scaled several levels further (see `eratosthenes` in Table 1). In cases where the sizes of the RF partitioning and the Shasha–Snir partitioning coincide (blue dots), the well-engineered Source outperforms our RF-SMC implementation. The time differences in these cases are reasonably moderate, suggesting that the polynomial overhead incurred to operate on the RF partitioning is small in practice.

Appendix C.4 of Bui et al. [2021] contains the complete results on all 109 benchmarks, as well as further scatter plots, illustrating (i) comparison of RF-SMC with Source and rfsc in SC, (ii) time comparison of RF-SMC across memory models, and (iii) the effect of using closure in the constency checking during SMC.

7 CONCLUSIONS

In this work we have solved the consistency verification problem under a reads-from map for the TSO and PSO relaxed memory models. Our algorithms scale as $O(k \cdot n^{k+1})$ for TSO, and as $O(k \cdot n^{k+1} \cdot \min(n^{k \cdot (k-1)}, 2^{k \cdot d}))$ for PSO, for n events, k threads and d variables. Thus, they both become polynomial-time for a bounded number of threads, similar to the case for SC that was established recently [Abdulla et al. 2019; Biswas and Enea 2019]. In practice, our algorithms perform much better than the standard baseline methods, offering significant scalability improvements. Encouraged by these scalability improvements, we have used these algorithms to develop, for the first time, SMC under TSO and PSO using the reads-from equivalence, as opposed to the standard Shasha–Snir equivalence. Our experiments show that the underlying reads-from partitioning is often much coarser than the Shasha–Snir partitioning, which yields a significant speedup in the model checking task.

We remark that our consistency-verification algorithms have direct applications beyond SMC. In particular, most predictive dynamic analyses solve a consistency-verification problem in order to infer whether an erroneous execution can be generated by a concurrent system (see, e.g., Kini et al. [2017]; Mathur et al. [2020]; Smaragdakis et al. [2012]). Hence, the results of this work allow to extend predictive analyses to TSO/PSO in a scalable way that does not sacrifice precision. We will pursue this direction in our future work.

ACKNOWLEDGMENTS

The research was partially funded by the ERC CoG 863818 (ForM-SMArt) and the Vienna Science and Technology Fund (WWTF) through project ICT15-003.

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction *(POPL)*. https://doi.org/10.1145/2578855.2535845

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *TACAS*. https://doi.org/10.1007/978-3-662-46681-0_28

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 150 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360576

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 135:1–135:29. https://doi.org/10.1145/3276505

S. V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: a tutorial. *Computer* 29, 12 (Dec 1996), 66–76. https://doi.org/10.1109/2.546611

Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. 2017. Context-Sensitive Dynamic Partial Order Reduction. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 526–543. https://doi.org/10.1007/978-3-319-63387-9_26

Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. 2018. Constrained Dynamic Partial Order Reduction. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 392–410. https://doi.org/10.1007/978-3-319-96142-2_24

Jade Alglave. 2010. *A Shared Memory Poetics*. Ph.D. Dissertation. Paris Diderot University.

Jade Alglave, Patrick Cousot, and Caterina Urban. 2017. Concurrency with Weak Memory Models (Dagstuhl Seminar 16471). *Dagstuhl Reports* 6, 11 (2017), 108–128. https://doi.org/10.4230/DagRep.6.11.108

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal Dynamic Partial Order Reduction with Observers. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 229–248. https://doi.org/10.1007/978-3-319-89963-3_14

Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28. https://doi.org/10.1145/3360591

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding Robustness against Total Store Ordering. In *Automata, Languages and Programming*, Luca Aceto, Monika Henzinger, and Jiří Sgall (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 428–440. https://doi.org/10.1007/978-3-642-22012-8_34

Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. 2021. The Reads-From Equivalence for the TSO and PSO Memory Models. *CoRR* abs/2011.11763 (2021). arXiv:2011.11763 https://arxiv.org/abs/2011.11763

Harold W. Cain and Mikko H. Lipasti. 2002. Verifying Sequential Consistency Using Vector Clocks. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Winnipeg, Manitoba, Canada) *(SPAA '02)*. Association for Computing Machinery, New York, NY, USA, 153–154. https://doi.org/10.1145/564870.564897

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158119

Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 124 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360550

Y. Chen, Yi Lv, W. Hu, T. Chen, Haihua Shen, Pengyu Wang, and Hong Pan. 2009. Fast complete memory consistency verification. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 381–392. https://doi.org/10.1109/HPCA.2009.4798276

Peter Chini and Prakash Saivasan. 2020. A Framework for Consistency Algorithms. In *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020, December 14-18, 2020, BITS Pilani, K K Birla Goa Campus, Goa, India (Virtual Conference) (LIPIcs, Vol. 182)*, Nitin Saxena and Sunil Simon (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:17. https://doi.org/10.4230/LIPIcs.FSTTCS.2020.42

E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. 1999. State space reduction using partial order techniques. *STTT* 2, 3 (1999), 279–287. https://doi.org/10.1007/s100090050035

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed Stateless Model Checking for SC and TSO *(OOPSLA)*. ACM, New York, NY, USA, 20–36. https://doi.org/10.1145/2814270.2814297

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *POPL*. https://doi.org/10.1145/1040305.1040315

Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. 2015. Memory-Model-Aware Testing: A Unified Complexity Analysis. *ACM Trans. Embed. Comput. Syst.* 14, 4, Article 63 (Sept. 2015), 25 pages. https://doi.org/10.1145/2753761

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (Aug. 1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

P. Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Secaucus, NJ, USA. https://doi.org/10.1007/3-540-60761-7

Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *POPL*. https://doi.org/10.1145/263699.263717

Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *FMSD* 26, 2 (2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li. 2012. Linear Time Memory Consistency Verification. *IEEE Trans. Comput.* 61, 4 (2012), 502–516. https://doi.org/10.1109/TC.2011.41

Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2737924.2737975

Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. *SIGPLAN Not.* 51, 10 (Oct. 2016), 447–461. https://doi.org/10.1145/3022671.2984025

Shiyou Huang and Jeff Huang. 2017. Speeding Up Maximal Causality Reduction with Static Dependency Analysis. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 16:1–16:22. https://doi.org/10.4230/LIPIcs.ECOOP.2017.16

Vineet Kahlon, Chao Wang, and Aarti Gupta. 2009. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 398–413. https://doi.org/10.1007/978-3-642-02658-4_31

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019a. Effective Lock Handling in Stateless Model Checking. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 173 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360599

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019b. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1157–1171. https://doi.org/10.1145/3373376.3378480

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. https://doi.org/10.1145/3062341.3062352

L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Magnus Lång and Konstantinos Sagonas. 2020. Parallel Graph-Based Stateless Model Checking. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12302)*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 377–393. https://doi.org/10.1007/978-3-030-59152-6_21

Tom Ball Madan Musuvathi, Shaz Qadeer. 2007. *CHESS: A systematic testing tool for concurrent software.* Technical Report.

C. Manovit and S. Hangal. 2006. Completely verifying memory consistency of test program executions. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* 166–175. https://doi.org/10.1109/HPCA.2006.1598123

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races *(POPL)*. https://doi.org/10.1145/3434317

Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 131–150. https://doi.org/10.1145/2509136.2509514

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371085

Doron Peled. 1993. All from One, One for All: On Model Checking Using Representatives. In *CAV*. https://doi.org/10.1007/3-540-56922-7_34

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31. https://doi.org/10.1145/3290382

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based Partial Order Reduction. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2015.456

Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: Efficient Predictive Race Detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 747–762. https://doi.org/10.1145/3385412.3385993

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

CORPORATE SPARC International, Inc. 1994. *The SPARC Architecture Manual (Version 9).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Rachid Zennou, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2019. Gradual Consistency Checking. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 267–285. https://doi.org/10.1007/978-3-030-25543-5_16

Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic Partial Order Reduction for Relaxed Memory Models. In *PLDI*. https://doi.org/10.1145/2737924.2737956