Truly Stateless, Optimal Dynamic Partial Order Reduction

MICHALIS KOKOLOGIANNAKIS, MPI-SWS, Germany
IASON MARMANIS, MPI-SWS, Germany
VLADIMIR GLADSTEIN, MPI-SWS, Germany and St Petersburg University/JetBrains Research, Russia
VIKTOR VAFEIADIS, MPI-SWS, Germany

Dynamic partial order reduction (DPOR) verifies concurrent programs by exploring all their interleavings up to some equivalence relation, such as the Mazurkiewicz trace equivalence. Doing so involves a complex trade-off between space and time. Existing DPOR algorithms are either exploration-optimal (i.e., explore exactly only interleaving per equivalence class) but may use exponential memory in the size of the program, or maintain polynomial memory consumption but potentially explore exponentially many redundant interleavings.

In this paper, we show that it is possible to have the best of both worlds: exploring exactly one interleaving per equivalence class with linear memory consumption. Our algorithm, TruSt, formalized in Coq, is applicable not only to sequential consistency, but also to any weak memory model that satisfies a few basic assumptions, including TSO, PSO, and RC11. In addition, TruSt is embarrassingly parallelizable: its different exploration options have no shared state, and can therefore be explored completely in parallel. Consequently, TruSt outperforms the state-of-the-art in terms of memory and/or time.

CCS Concepts: • **Theory of computation → Concurrency**; **Verification by model checking**.

Additional Key Words and Phrases: Model Checking, Dynamic Partial Order Reduction, Weak Memory Models

ACM Reference Format:
Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly Stateless, Optimal Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 6, POPL, Article 49 (January 2022), 28 pages. https://doi.org/10.1145/3498711

1 INTRODUCTION

Stateless model checking (SMC) [Godefroid 1997; Musuvathi et al. 2008] is an effective verification technique for verifying concurrent programs of bounded size. It was introduced as an alternative to explicit-state model checking that avoids excessive memory consumption and therefore has the potential to scale to larger programs. SMC works by systematically exploring all executions of a given concurrent program without ever storing the set of program states it has already visited.

While SMC allows for a program to be verified with polynomial memory requirements, it has the obvious downside that the number of executions to be explored is typically exponential in the size of the program. Thus, SMC is almost always employed together with clever *dynamic partial order reduction* (DPOR) algorithms [Abdulla et al. 2014; Flanagan et al. 2005; Kokologiannakis et al. 2019] that reduce the number of executions that need to be explored in order to cover all possible program behaviors. DPOR partitions the execution traces of a program into *equivalence classes* according to some relation, such as Mazurkiewicz trace equivalence [Mazurkiewicz 1987], with the

Authors' addresses: Michalis Kokologiannakis, MPI-SWS, Saarland Informatics Campus, Germany, michalis@mpi-sws.org; Iason Marmanis, MPI-SWS, Saarland Informatics Campus, Germany, imarmanis@mpi-sws.org; Vladimir Gladstein, MPI-SWS, Germany and St Petersburg University/JetBrains Research, Russia, vgladstein@mpi-sws.org; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2022/1-ART49
https://doi.org/10.1145/3498711

property that all equivalent traces exhibit the same observable outcome. Then, to verify a program, it suffices to explore one trace from each equivalence class.

Exploring one representative execution trace from each equivalence class (i.e., being *optimal*), however, is not easy. DPOR algorithms (e.g., [Abdulla et al. 2015; 2014; 2016; Albert et al. 2017; 2018; Chalupa et al. 2017; Chatterjee et al. 2019; Flanagan et al. 2005; Kokologiannakis et al. 2019; Zhang et al. 2015]) typically start by exploring one program trace, and whenever they detect a racy pair of accesses, they explore additional traces that contain the racy accesses in reverse order, while also maintaining some state to avoid re-exploring an equivalent execution trace. Existing algorithms, however, either are nonoptimal, which means that they may explore an exponential number of traces even for programs with $O(n)$ equivalence classes (where n is the size of the program), or achieve optimality by sacrificing the very thing SMC was invented for: polynomial memory consumption. In fact, most modern DPOR solutions to the verification problem opt for the second solution, i.e., they may consume an exponential amount of memory.

In this paper, we show that such a compromise is unnecessary. We develop an *exploration-optimal* DPOR algorithm with *linear memory requirements*. Besides providing a scalable solution for DPOR in terms of both time and memory, our algorithm, TruSt (Truly Stateless Model-checker), has two other big advantages:

- It is *parametric* both in (1) the choice of the memory model, supporting not only sequential consistency (SC) [Lamport 1979] but also a wide range of weak memory models, such as TSO [Sewell et al. 2010], PSO [SPARC International Inc. 1994], and RC11 [Lahav et al. 2017], and also in (2) the choice of the DPOR equivalence relation, supporting both Shasha-Snir equivalence [Shasha et al. 1988] (the generalization of Mazurkiewicz equivalence to weak memory models) and reads-from equivalence [Chalupa et al. 2017].
- Its explorations of different execution traces share absolutely no state, which means that TruSt is embarrassingly *parallelizable*. This is in contrast to existing DPOR solutions, whose parallelization requires sharing to avoid duplication (e.g., [Lång et al. 2020]).

To achieve this, we build upon prior work—most notably, GenMC [Kokologiannakis et al. 2019]—and represent program executions as *execution graphs* [Alglave et al. 2014]. However, we radically change the conditions under which racy accesses are reversed by DPOR. Specifically, we restrict reversals to happen only when the events to be removed form a *maximal extension* of the remaining execution—a novel notion that we introduce in this paper. We show that maximal extensions always exist and are unique, and so one can achieve correctness and optimality without recording any additional state. Linear space complexity stems from the fact that TruSt explores executions in a recursive depth-first manner: the recursive depth is bounded by the size of the program, and each recursive call requires constant space to represent its difference from the current execution graph, which is also linear in size.

In summary, we make the following contributions:

§2 Through a series of examples, we describe how existing DPOR solutions work and why maintaining both optimality and polynomial space complexity poses a considerable challenge.
§3 We provide an intuitive account of how TruSt surmounts this challenge by representing program executions as graphs and restricting alternative explorations using a notion of maximality. Our algorithm works for any memory model subject to a few basic assumptions.
§4 We describe our algorithm in detail, and *prove* (in Coq) that it is sound, complete and optimal.
§5 We implement a parallel version of TruSt into a tool for verifying C/C++ programs.
§6 We demonstrate that TruSt outperforms the state-of-the-art in terms of memory and/or verification time, and that it scales very well on multicore machines.

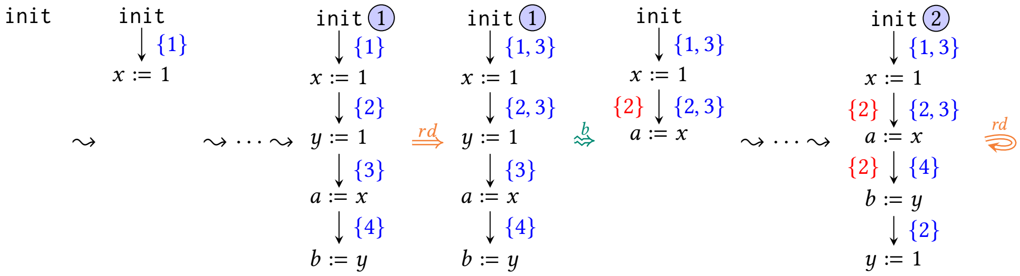

Fig. 1. DPOR exploration of the first and second trace of W+W+RR

2 SMC & DPOR: A TROUBLED MARRIAGE

Why is it that SMC/DPOR suffers from this exploration/memory trade-off, to begin with? To answer this question, let us start by recalling the fundamentals of DPOR. To ease the presentation, in the rest of this section we assume a setting of sequential consistency (SC), and that the notion of equivalence used is that of Mazurkiewicz equivalence. In §3, we lift these assumptions and arrive at TruSt, a memory-model agnostic DPOR that works both for Mazurkiewicz-like equivalence classes, as well as coarser equivalence partitionings.

2.1 DPOR in a Nutshell

As briefly mentioned, DPOR first explores one full program interleaving and then goes into a *race-detection* phase, when it identifies further interleavings that need to be explored. Specifically, it detects conflicting transitions, which, if executed in the reverse order, will lead to interleavings belonging to a different equivalence class. DPOR then explores these alternatives one by one in a depth-first way. (After each interleaving is explored, a race-detection phase is run, which can recursively generate further interleavings to be explored and so on.)

We illustrate the above procedure with the W+W+RR example below, where each instruction is given a distinct label.

$$(1)\ x := 1; \;\Big\|\; (2)\ y := 1; \;\Big\|\; \begin{array}{l}(3)\ a := x;\\ (4)\ b := y\end{array} \qquad (\text{W+W+RR})$$

This program has 12 $(\frac{4!}{2!})$ interleavings, grouped into 4 Mazurkiewicz equivalence classes.

Let us now see how DPOR generates all four Mazurkiewicz traces of this program. Starting with an empty trace, DPOR adds events corresponding to the transitions of the program until it arrives at the full trace ①, as can be seen in Fig. 1. While doing so, it keeps track of the transition executed at each step in a *backtrack set*, in which it will later record any alternative exploration options.

Once the first trace is fully explored, DPOR initiates a race-detection ($\xRightarrow{rd}$) phase, and looks for conflicting transitions. Under Mazurkiewicz equivalence, two transitions are *conflicting* if they access the same memory location, and at least one of them is a write instruction. Whenever DPOR detects two conflicting transitions, it populates the backtrack set of the earlier transition with another available transition, which will force it to consider the racy instructions in reverse order.

Returning to the W+W+RR example, there are two pairs of conflicting transitions: $\langle 1, 3\rangle$ and $\langle 2, 4\rangle$. For the former pair, DPOR populates the backtrack set of the initial state with (3), which can be executed instead of the first transition of the trace. For the latter pair, however, note that (4) cannot be executed instead of transition (2) because (3) has to be executed first. For this reason, DPOR populates the backtrack set of the second transition with (3) instead of (4). (In general, when the

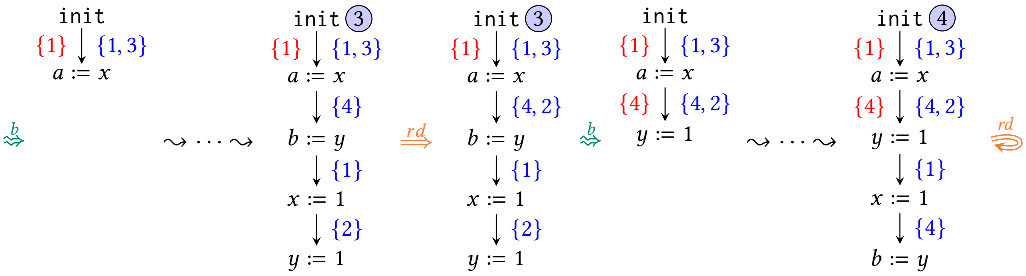

Fig. 2. DPOR exploration of the third and fourth trace of W+W+RR

transition to be recorded in a backtrack set cannot be immediately executed because it causally depends on some previous set of events E, DPOR instead records some other event in the backtrack set—typically, but not necessarily, one of the E events.)

Now that the race-detection phase is complete, DPOR considers alternative exploration options in a depth-first fashion. It locates the latest transition for which there are unexplored transitions in its backtrack set and backtracks ($\overset{b}{\rightsquigarrow}$) by restricting the trace to include only the events fired before that transition, and then firing a transition that has not been explored. For W+W+RR the first such transition is the second one, and thus DPOR first tries to reverse the race between (2) and (4). To do that, however, (2) should not be executed until the race is reversed (i.e., until after (4) is executed). DPOR achieves this by putting (2) in the *sleep set*, and then executes the remaining transitions of the program, starting with (3), as indicated by the backtrack set. As can be seen in Fig. 1, (2) is removed from the sleep set upon executing (4) (or, more generally, any conflicting transition).

At this point, the second full trace (trace ②) is constructed, and DPOR again initiates a race-detection phase. During this phase, two races are detected: one between transitions (1) and (3), and one between (2) and (4). However, neither of these races will have any effect on the backtrack sets. As far as the first race is concerned, this is pretty much expected, since (3) is already in the backtrack set of the initial transition there. Perhaps surprisingly, however, DPOR will also not populate the backtrack set of the third transition, despite the fact that (2) does not belong to the backtrack set there. The reason for that is that (2) is in the sleep set of that transition. This indicates that this would "re-reverse" a race that is being reversed (leading to a trace equivalent to trace ①), and thus DPOR avoids attempting to reverse the second race altogether.

Subsequently, DPOR backtracks further (cf. Fig. 2) and explores alternative options for the first transition. Since firing (1) for this transition has already been explored, (1) is inserted into the sleep set, and (3) is fired instead. At this point, (1) is immediately removed from the sleep set, as it is in a race with (3). Then, DPOR can continue with either of (1), (2) and (4), but let's assume it first chooses (4), and then continues with (1) and (2), eventually leading to trace ③.

Continuing with the race-detection phase, which should by now be familiar, DPOR notices that (2) is in a data race with (4), and thus inserts (2) in the backtrack set of the second transition.

Finally, the algorithm backtracks to the second transition and fires (2) instead of (4). It then adds (1) and (4) yielding trace ④. As before, the race-detection phase of ④ does not add any new transitions to the backtrack sets, thereby concluding the exploration of representative interleavings from all four distinct Mazurkiewicz equivalence classes of the program.

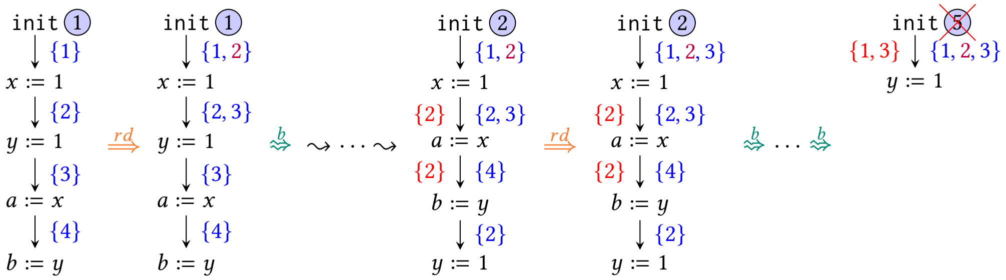

Fig. 3. An unlucky DPOR exploration for w+w+rr, leading to a blocked execution

2.2 The Race-Reversal Problem

DPOR, as presented so far, has one big disadvantage: the way a race is reversed is not predetermined. When DPOR detects a race between two events a and b with a appearing before b in the trace, it needs to populate the backtrack set at a's position. If, however, b is not fireable at a's position (e.g., because it causally depends on some other events that were added after a[1]), a different event is added instead. In fact, we have already seen an example of this issue: during the race-detection phase of the first trace of w+w+rr, (3) was added at the backtrack set of (2) instead of (4), as the latter was not fireable at that point, due to it depending on transition (3).

In general, DPOR populates the backtrack set at a with an event c that appears between a and b in the trace, and does not causally depend on any other events. Of course, while it does make sense to add b itself (or a predecessor of b from the same thread) to a's backtrack set, this is not always possible, as b might depend on multiple other events, spanning many different threads.

This non-determinism in the race-reversal mechanism occasionally leads DPOR into encountering blocked executions. To see an example of this, consider again the w+w+rr example from §2.1, but this time suppose that during the race-detection phase of trace ①, (2) is added to the backtrack set of the first transition (instead of (3)), as shown in Fig. 3. During the race-detection of trace ②, DPOR will discover the same races as in the exploration of §2.1. This time, however, there is a key difference: the race between (1) and (3) *has* to be reversed. In the run of §2.1, (3) was already in the backtrack set of the first transition, so during the race-detection phase of trace ②, DPOR did not alter the backtrack set of the first transition. In the current exploration, however, (3) does not belong to the backtrack set of the first transition. Thus, DPOR adds it there, leading to a backtrack set with three events.

In turn, while the exploration of traces ②, ③ and ④ proceeds exactly as in §2.1, the different backtrack set of the first transition will lead to an extraneous exploration. Indeed, as can be seen in Fig. 3, DPOR starts an exploration by firing (2) as the first transition. All other fireable transitions, however, are in the sleep set, which means that this extraneous exploration is going to be blocked.

More generally, even though in this example the basic DPOR algorithm explores only one blocked execution, there are parametric programs with a linear number of Mazurkiewicz equivalence classes, where DPOR explores exponentially many blocked executions [Abdulla et al. 2017; Nguyen et al. 2018]. Solving the race-reversal problem described above in an optimal manner (i.e., exploring exactly one trace per Mazurkiewicz equivalence class) is surprisingly difficult. And, even though there are DPOR algorithms that manage to achieve such optimality [Abdulla et al. 2017; 2019; 2018; Aronis et al. 2018; Kokologiannakis et al. 2019; 2020], as we are going to shortly see, all such solutions suffer from exponential memory consumption.

[1] The exact notion of "causality" depends on the particular DPOR used.

2.3 Exponential-Cost Solutions to the Race-Reversal Problem

But how do existing DPOR solutions actually achieve optimality? The key idea behind these techniques is to save not mere transitions in the backtrack set, but rather *transition sequences*. Thus, whenever DPOR tries to reverse a race between two events a and b (with a being earlier in the trace), it does not have to "guess" which transition it should add to the backtrack set at a's position; instead, it can simply add a sequence comprising b and all its causal predecessors, so that the race between a and b is precisely reversed. By doing that, it can also get rid of the sleep set, which is only used to not fire a transition prematurely (i.e., before the race is reversed).

As a concrete example, let us briefly discuss how the exploration procedure of the optimal-DPOR algorithm of Abdulla et al. [2014] would differ when verifying the w+w+rr example. During the race-detection phase of trace ①, optimal-DPOR would add the sequences 3 and 3.4 to reverse the races between (1) and (3), and (2) and (4), respectively. By doing so, optimal-DPOR can precisely reverse all races in the program and also avoid exploring any blocked executions.

While saving transition sequences is sufficient to guarantee optimality, it can lead to exponential memory consumption [Abdulla et al. 2014; Nguyen et al. 2018]. To see an example of this, consider the exp-mem program below.

$$\texttt{fetch\_add}(x,1) \;\Bigg\|\; r_1 := \texttt{fetch\_add}(y,1) \;\Big\|\; \ldots \;\Big\|\; \begin{array}{l} r_N := \texttt{fetch\_add}(y,1) \\ \texttt{fetch\_add}(x,1) \end{array} \qquad (\textsc{exp-mem})$$

The problem with this program is that an exponential number of transition sequences will be stored in the backtrack set of the first x access, and these will not be explored until all races on the y accesses are reversed. Concretely, assume that the optimal-DPOR algorithm obtains the first trace of the program by executing the program threads in a left-to-right order. During the race-detection phase of that trace, the backtrack set of the first x access will be populated with a transition sequence s, due to the race between the two x accesses. Since the second x access causally depends on all the preceding accesses to y, s will contain *all* transitions fired after the first access to x.

Of course, the size of s is not a problem; what is, however, a problem is that DPOR must also reverse all the races on y. And, what is worse, is that, for each of these reversals, another (new) transition sequence will be inserted to the backtrack set of the first x access, as it will differ from all the existing ones in the ordering of the y accesses. Given that there are $N!$ ways the y accesses can be ordered, and that all these $N!$ orderings will have to be explored before the race on x is reversed (due to the DFS-like nature of DPOR), it becomes clear that the backtrack set of the first x access will consume an exponential amount of memory.

3 TruSt: RECONCILING SMC & DPOR

In this section, we describe TruSt, our DPOR framework that combines three key features: (1) memory-model parametricity, (2) optimality, and (3) polynomial memory requirements. In the following subsections, we progressively describe how TruSt achieves each of these features. To avoid confusion, we use a different index (i.e., TruSt$_0$, TruSt$_1$) for each of the "intermediate" versions of TruSt, until we arrive at our full algorithm in §3.5.

3.1 TruSt$_0$: Representing Executions as Graphs

In order to support weak memory models, we cannot simply represent program executions as interleavings, as we have to take into account the possible reorderings allowed by such models. We therefore represent the executions of a concurrent program as *execution graphs* [Alglave et al. 2014], comprising a set of events, and a few relations on them.

Definition 3.1. An *event*, $e \in \mathsf{Event}$, is either the initialization event $\mathtt{init}$, or a thread event $\langle t, i, lab\rangle$ where $t \in \mathsf{Tid}$ is a thread identifier, $i \in \mathsf{Idx} \triangleq \mathbb{N}$ is a serial number inside each thread, and $lab \in \mathsf{Lab}$ is a label that takes one of the following forms:

- Write label: $\mathtt{W}^k(l, v)$ where $k \in \mathsf{Kind}$ is the kind of the write (e.g., normal, exclusive) depending on the programming language, $l \in \mathsf{Loc}$ is the location accessed, and $v \in \mathsf{Val}$ the value written.
- Read label: $\mathtt{R}^k(l)$ where $k \in \mathsf{Kind}$ is the kind of the read and $l \in \mathsf{Loc}$ is the location accessed.
- Fence label: $\mathtt{F}^k$ where $k \in \mathsf{Kind}$ is the kind of the fence (e.g., full-fence, store-fence).
- Error label: $\mathtt{error}$.

When applicable, the functions $\mathtt{tid}$, $\mathtt{idx}$, $\mathtt{loc}$, and $\mathtt{val}$, return the thread identifier, serial number, location, and value of an event, respectively. We use $\mathsf{R} \triangleq \{\langle t, i, lab\rangle \mid lab = \mathtt{R}(\_)\}$ to denote the set of all read events, $\mathtt{W} \triangleq \{\mathtt{init}\} \cup \{\langle t, i, lab\rangle \mid lab = \mathtt{W}(\_, \_)\}$ to denote the set of all write events (which includes the initialization event), and $\mathtt{error} \triangleq \{\langle t, i, lab\rangle \mid lab = \mathtt{error}\}$ to denote the set of all error events. We use subscripts to further restrict those sets (e.g., $\mathtt{W}_l \triangleq \{\mathtt{init}\} \cup \{w \in \mathtt{W} \mid \mathtt{loc}(w) = l\}$).

Definition 3.2. An *execution graph* G consists of:

(1) a set $G.\mathsf{E}$ of events that includes $\mathtt{init}$ and does not contain multiple events with the same thread identifier and serial number.
(2) a total order $\leq_G$ on $G.\mathsf{E}$, representing the order in which events were incrementally added to the graph by the TruSt algorithm,
(3) a function $G.\mathtt{rf} : G.\mathsf{R} \rightarrow G.\mathtt{W}$, called the *reads-from* function, that maps each read event to a same-location write from where it gets its value, and
(4) a strict partial order $G.\mathtt{co} \subseteq \bigcup_{l \in \mathsf{Loc}} G.\mathtt{W}_l \times G.\mathtt{W}_l$, called the *coherence order*, which is total on $G.\mathtt{W}_l$ for every location $l \in \mathsf{Loc}$,

writing $G.\mathsf{R}$ for the set $G.\mathsf{E} \cap \mathsf{R}$ and similarly for other sets. Given two events $e_1, e_2 \in G.\mathsf{E}$, we write $e_1 <_G e_2$ if $e_1 \leq_G e_2$ and $e_1 \neq e_2$. We write $G|_E$ for the restriction of an execution graph G to a set of events E, and $G \setminus E$ for the graph obtained by removing a set of events E. Finally, we write $G_1 \approx G_2$ if the two graphs are equal up to the the $\leq_G$ component (i.e., they agree on all other components).

Our definition of execution graphs differs from most presentations in the literature in two ways. First, it contains one additional component: the total order $\leq_G$, which is used by TruSt to record the order in which events were added to an execution. Second, G does not have an explicit *program order* (po) component. Instead, po is induced by the representation of events as a partial order that orders events of the same thread according to their serial numbers and initialization events before all non-initialization events.

$$\mathsf{po} \triangleq \left\{\langle \mathtt{init}, e\rangle \;\middle|\; e \in \mathsf{Event} \setminus \{\mathtt{init}\}\right\} \cup \left\{\langle\langle t_1, i_1, lab_1\rangle, \langle t_2, i_2, lab_2\rangle\rangle \;\middle|\; t_1 = t_2 \land i_1 < i_2\right\}$$

We define the causal dependency relation $G.\mathsf{po}\mathtt{rf}$ as the transitive closure of the program order and the reads-from dependencies.

$$G.\mathsf{po}\mathtt{rf} \triangleq (\mathsf{po} \cap (G.\mathsf{E} \times G.\mathsf{E}) \cup \{\langle G.\mathtt{rf}(r), r\rangle \mid r \in G.\mathsf{R}\})^+$$

The semantics of a program P under a memory model m is then given by the set of execution graphs corresponding to the program that satisfy the consistency predicate of m. Consistency typically ensures that reads return relatively recent values, and that the coherence ordering of writes does not contradict the program order.

As an example, the consistent execution graphs of W+W+RR under SC are depicted in Fig. 4. When displaying graphs, we follow the standard convention in the weak memory literature and draw the $\mathtt{rf}$ function as a dashed arrow indicating the data flow from the write to the read (as opposed to the functional mapping from each read to the corresponding write). Similarly, we draw

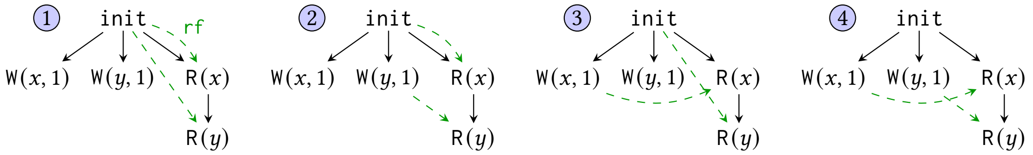

Fig. 4. Execution graphs of W+W+RR under SC

only the non-transitive po edges as solid black arrows, and leave the t and i components of events implicit. Finally, we typically do not draw co-edges from the initialization event to other writes.

Observe that execution graphs subsume the notion of "equivalence classes" used by DPOR. Indeed, as can be seen in Fig. 4, W+W+RR has 4 consistent execution graphs under SC, even though it has 12 interleavings. Although the precise equivalence partitioning induced by these graphs depends on the memory model definition and on the relations recorded, observe that actions performed by different threads are not ordered among themselves, and that accesses to different variables are also unordered by default.

TruSt can be instantiated for any memory model m that satisfies three basic assumptions.[2]

Well-formedness: Consistency does not depend on the order in which events are added to the graph (i.e., if G is consistent, then so is any graph $G' \approx G$), and, in consistent graphs, porf should be acyclic (i.e., an event cannot circularly depend on itself).

Prefix-closedness: Restricting a consistent graph to any porf-prefix-closed subset of its events yields a consistent graph. Prefix-closedness enables TruSt to construct a consistent graph incrementally.

Maximal-extensibility: Adding a po-maximal event to a consistent graph preserves consistency if added in a maximal way: co-maximally for writes, and reading from the co-maximal write for reads. Intuitively, executing a program should never get stuck if a thread has more statements to execute: the remaining statements can always be executed with SC semantics (in particular, each read can return the value written by the most recent, same-location write).

3.2 TruSt$_0$: Partially Alleviating the Race-Reversal Problem

Execution graphs enable TruSt$_0$ to perform two key optimizations which together allow it to reverse over 50% of the races optimally.

The first optimization stems from the observation that DPOR's race-detection phase need not take place at the end of an execution but can be executed incrementally each time a new event is added to the graph. Thus, whenever an event b is added to the graph, it suffices to check whether it conflicts with some event a already in the graph G. Given the graph representation, such a check is very natural because if b is a write or a read event, TruSt$_0$ would anyway have to determine the last event same-location write in G so as to update the co or the rf components respectively.

The second optimization exploits the semantics of writes when reversing races: namely, that writes only affect the values that can be read from memory, and not the local state of any program thread. Therefore, if we have a race between a write event a and a later event b, we do not need to remove any events from the execution graph. If b is a read event, it suffices to change $G.\mathsf{rf}(b)$; while if b is a write event, it suffices to change the co edge between a and b while keeping other events intact.

[2] The first two conditions are identical to those of Kokologiannakis et al. [2019]. Our third condition (maximal extensibility) is slightly stronger than their corresponding condition but still satisfied by all standard memory models.

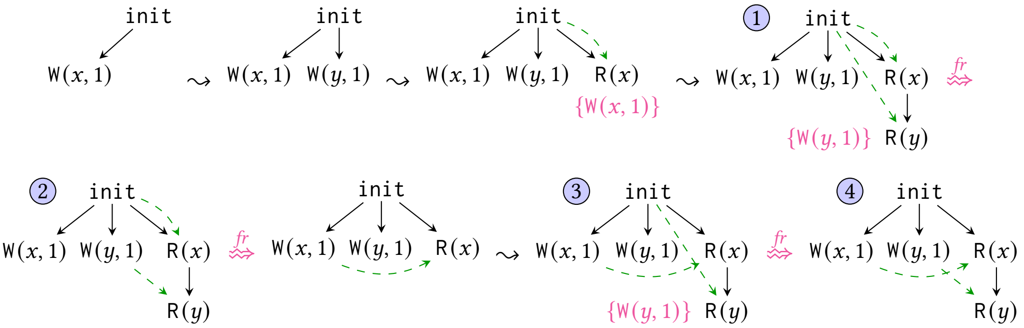

Fig. 5. DPOR exploration of the execution graphs of W+W+RR

We illustrate these optimizations on the W+W+RR example (cf. Fig. 5). Similarly to an interleaving-based DPOR, TruSt_0 starts with an initial graph and adds events corresponding to the instructions of the program one at a time. In contrast to an interleaving-based DPOR, however, when TruSt_0 adds a read event, its rf options are not limited to a single write; instead, they are determined by the memory model's consistency predicate. For instance, when TruSt_0 adds $\mathtt{R}(x)$, it is consistent for it to read both 0 and 1. Thus, TruSt_0 proceeds with one rf option (e.g., 0) and records the other option in a *revisit set* associated with the read. (The revisit set is somewhat analogous the backtracking set of §2, but note that it associates the alternative option with the event added last to the graph.) In a similar manner, when $\mathtt{R}(y)$ is added, it will read one possible value (e.g., 0), while any alternatives (in this case, 1) will be added to its revisit set.

Since the first graph (graph 1) is complete, TruSt_0 backtracks and explores any recorded alternative options in a depth-first fashion. The backtracking procedure of TruSt_0, however, differs from that of an interleaving-based DPOR. When backtracking in order for $\mathtt{R}(y)$ to read from $\mathtt{W}(y, 1)$, instead of returning to the initial state (as an interleaving-based DPOR would do), TruSt_0 backtracks to the point $\mathtt{R}(y)$ was added, and simply changes its rf so that it reads from $\mathtt{W}(y, 1)$, immediately yielding execution 2. To distinguish between the backtracking procedure of the interleaving-based DPOR and that of TruSt_0, we call TruSt_0's backtracking *revisiting* and this kind of optimized revisiting that does not remove any events that were added in between the two racy events a *forward revisit*.

The rest of the exploration proceeds in a similar manner. Once execution 2 is complete, TruSt_0 revisits $\mathtt{R}(x)$ and changes its rf to $\mathtt{W}(x, 1)$, and then adds $\mathtt{R}(y)$ and $\mathtt{W}(y, 1)$ again, yielding execution 3, and through another forward revisit of $\mathtt{R}(y)$ also execution 4.

The reason TruSt_0 is able to backtrack in this optimized manner is attributed to the way it checks consistency. Indeed, precisely because consistency does *not* depend on the order of the different events in a trace, TruSt_0 is able to keep both the conflicting read and write in the graph, and "reverse" the race merely by changing the read's rf edge. By contrast, an interleaving-based DPOR always has to remove both conflicting events from the trace (as well as the causal predecessors of the event later in the trace), so that they can be later re-added in the correct order.

3.3 TruSt_0: The Backward Revisit Problem

We have just seen how TruSt_0 resolves the race-reversal problem for forward revisits by updating the execution graph in place without removing any intermediate events. This approach, however, does not work for *backward revisits*, namely ones where the earlier event is a read and the later event is a write, because the events that were added to the graph between the read and the write

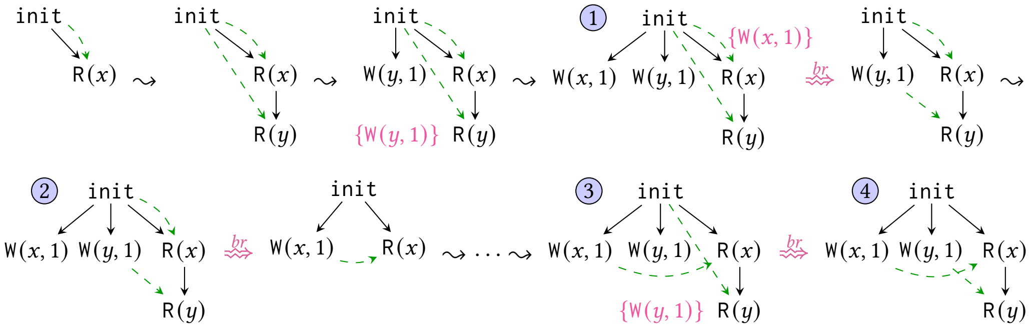

Fig. 6. DPOR exploration of the execution graphs of w+w+rr (right-to-left order)

may depend on the value read and thus need to be removed. To ensure that the write and its causal (porf) predecessors will always be added, similar to optimal-DPOR from §2.3, TruSt$_0$ stores the porf-predecessors of the revisiting write in the read's revisit set.

We illustrate TruSt$_0$'s handling of backward revisits again with the w+w+rr example, but now assuming that the program events are added in a right-to-left-order (cf. Fig. 6). As it can be seen, TruSt$_0$ adds events one by one, until it eventually arrives at execution ①. When $\mathtt{W}(y, 1)$ is added, TruSt$_0$ notices that $\mathtt{R}(y)$ can be revisited to read from it, and thus adds an item to the revisit set of $\mathtt{R}(y)$, containing the causal prefix of $\mathtt{W}(y, 1)$. Analogously, when $\mathtt{W}(x, 1)$ is added, TruSt$_0$ adds an item with the causal prefix of $\mathtt{W}(x, 1)$ to the revisit set of $\mathtt{R}(x)$.

Having completed a full execution, TruSt$_0$ explores the recorded revisits in a depth-first fashion. It therefore starts with the backward revisit of $\mathtt{R}(y)$ and so it restricts the graph to contain only the events that were added before $\mathtt{R}(y)$, as well as the porf-prefix of $\mathtt{W}(y, 1)$, which is $\mathtt{W}(y, 1)$ itself.

It then adds the $\mathtt{W}(x, 1)$ event completing execution ②. Note that, at this point, nothing needs to be recorded in the revisit set of $\mathtt{R}(x)$, as it already contains $\mathtt{W}(x, 1)$.

Next, the algorithm proceeds with backward-revisiting $\mathtt{R}(x)$. The restricted graph now contains only $\mathtt{R}(x)$, which was the first event, and $\mathtt{W}(x, 1)$, which does not have any other causal predecessors. It then adds $\mathtt{R}(y)$, which can only read 0, and then $\mathtt{W}(y, 1)$, which completes execution ③ and backward-revisits $\mathtt{R}(y)$ leading to execution ④.

Intuitively, we can imagine the graph induced by the backward revisit of a read r from a write w, as the forward revisit of r that would occur if w's prefix was already present in the graph when r was added. In other words, if we had added the events that are necessary to trigger w before adding r, then r could have also read from w. As a concrete example of this, when $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$ after execution ② in Fig. 6, the graph we obtain from the backward revisit effectively models the scenario where $\mathtt{W}(x, 1)$ was already present when $\mathtt{R}(x)$ was added.

While saving prefixes avoids constructs like sleep sets for TruSt$_0$, alone it is not enough to eliminate the need for revisit sets. To see this, consider the backward revisits that can be performed by $\mathtt{W}(x, 1)$ in the steps leading to executions ① and ②. In both cases, $\mathtt{W}(x, 1)$ can revisit $\mathtt{R}(x)$ and lead to the same graph. Performing the same backward revisit twice, however, would obviously lead to duplication. Revisit sets are necessary to preclude such duplication.

Remark. In fact, no item of a revisit set may be removed from the set until the corresponding read is deleted from the graph, as shown by the example in Fig. 7, where for conciseness, we do not record co between the two writes to x. Assuming that TruSt$_0$ first explores the backward revisit from $\mathtt{W}(x, 1)$ and then the one from $\mathtt{W}(x, 2)$ after execution ①, when $\mathtt{W}(x, 2)$ is re-added in execution

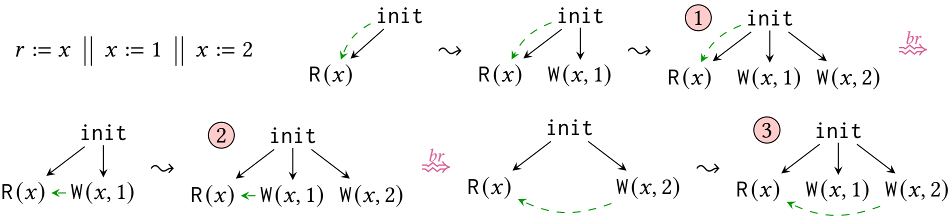

Fig. 7. TruSt$_0$: Why items cannot be removed from revisit sets immediately upon exploration

③, we have to remember that it has revisited $\mathtt{R}(x)$ in the past. Thus, we cannot remove items from the revisit set of a particular read once these items have been explored.

3.4 TruSt$_1$: Achieving Optimality via Maximal Extensions

TruSt avoids the need for revisit sets by ensuring that a particular backward revisit can only occur once among all graphs, under a novel condition we call *maximal extension* condition, which requires that all events affected (e.g., removed) by the revisit must have been added in a co-maximal fashion. The maximal extension condition allows us to completely eliminate revisit sets, and can be considered TruSt's cornerstone: as we will see in § 3.5, maximal paths are the reason why TruSt achieves polynomial memory requirements. Let us now incorporate the maximal extension condition into TruSt$_1$.

The key idea behind maximal extension is simple. Consider the (consistent) graph G' that may occur as a result of a backward revisit of a read event r by a write event w. From prefix-closedness (see §3.1), we also know that $G'' \triangleq G' \setminus \{r, w\}$ is consistent. Starting from G'' and assuming a fixed construction order, if we add all the remaining events of the program, we can in general arrive to multiple graphs $G_1, G_2, \ldots$, depending on the way we add the remaining events (e.g., the rf edges of reads, etc). In principle, these are all the graphs which can lead to a backward revisit of r from w. Our goal is to both allow such revisiting in only one of these graphs, G, and also ensure that such a graph exists. We achieve this by requiring that (1) all additional events in G be added in a co-maximal manner, and (2) no revisiting takes place while constructing G. Uniqueness follows because there is only one choice of an rf/co extension that would make a given event co-maximal when no revisiting takes place, while existence follows from maximal extensibility (see §3.1): since G'' is consistent, it is always consistent to add events in a co-maximal manner.

Let us now formalize this intuition. We say that a write event $w \in G.\mathtt{W}$ is co-*maximal* w.r.t. a set of events E if $w \in E$ and there is no $w' \in E$ such that $\langle w, w' \rangle \in G.\mathtt{co}$. A read event $r \in G.\mathtt{R}$ is co-*maximal* w.r.t. E if $G.\mathtt{rf}(r)$ is co-maximal w.r.t. E. An event $e \in G.\mathtt{E}$ is *maximally added* before a write event $w \in G.\mathtt{W}$ if e is co-*maximal* w.r.t. the set $\mathit{Previous}_G(e, w) \triangleq \{e' \in G.\mathtt{E} \mid e' \leq_G e \vee \langle e', w \rangle \in G.\mathtt{porf}\}$, and there does not exist $r \in \mathit{Previous}_G(e, w)$ such that $G.\mathtt{rf}(r) = e$.

Definition 3.3 (Maximal Extension). An execution graph G is a *maximal extension* of a potential backward revisit from $w \in G.\mathtt{W}$ to $r \in G.\mathtt{R}$ if every $e \in G.\mathtt{E}$ such that $r \leq_G e$ and $\langle e, w \rangle \notin G.\mathtt{porf}$ is added maximally before w.

The above definition closely follows the intuitive description above, so let us go through it in detail, while keeping the above explanation in mind. First, notice that co-maximality of an event e is checked w.r.t. the set $\mathit{Previous}_G(e, w)$, which contains e and all events added before it, as well as those events that are (strictly) porf-before w, since the latter events will be included in the resulting graph G'. Second, notice that co-maximality is only required for r and all the events added after it, excluding those events that are strictly porf-before of w (that is, including w itself). The reason why the strict porf-prefix of w is excluded is because, as explained previously, this prefix will be included in the resulting graph G'. The reason why w is included, on the other hand, is that, when starting from G'', we can generally add w in multiple different ways as far as its co position w.r.t. the events that are going to be deleted is concerned, and thus we have to pick one. Finally, notice that the definition of $\mathit{Previous}_G(e, w)$ forbids backward revisits from deleted events.

Let us now see an example of how TruSt_1 avoids considering the same backward revisit twice, again using the W+W+RR example and Fig. 6. As already explained, the backward revisit of $\mathtt{R}(x)$ from $\mathtt{W}(x, 1)$ is examined twice in executions ① and ②. However, according to Def. 3.3, the backward revisit will only be considered in execution ①, since $\mathtt{R}(y)$, $\mathtt{W}(y, 1)$ and $\mathtt{W}(x, 1)$ were all added in a co-maximal manner[3]. (Note that $\mathtt{R}(y)$ is not reading from $\mathtt{W}(y, 1)$, which is the co-maximal write in execution ①, but that is OK, since we only want events to be maximal when they are added.) By contrast, graph ② is not a maximal extension of the same revisit because $\mathtt{R}(y)$ was not maximally added: it is reading from $\mathtt{W}(y, 1)$, which was added to the graph after it and does not belong to the porf-prefix of $\mathtt{W}(x, 1)$.

Another way of seeing why TruSt_1 should *not* do the revisit of $\mathtt{R}(x)$ in execution ②, is that, by doing it, it would "undo" the previous revisit of $\mathtt{R}(y)$ by $\mathtt{W}(y, 1)$. This is indeed the case: the maximal extension constraint ensures that a backward revisit cannot be contained among the events that will be deleted by a subsequent backward revisit.

As we will shortly see (§3.5), the fact that TruSt_1 avoids undoing work that it has already done, along with the fact that revisit sets are effectively transformed into revisit lists (since TruSt_1 no longer has to keep prefixes around), are crucial in achieving polynomial memory requirements in the final version of TruSt.

3.5 TruSt: From Exponential to Linear Memory Requirements

Even though maximal paths render revisit sets obsolete (in the sense that TruSt_1 does not need to remember the sequences that have backward-revisited each read), TruSt_1 still stores sequences that can revisit each read as part of a revisit list. However, as demonstrated by EXP-MEM in §2.3, similarly to backtrack sets, such lists may grow to be exponentially large before we start removing items from them, even if removing items is actually possible.

Let us now see how the final version of TruSt extends TruSt_1 to solve the above problem. TruSt's solution is to explore all revisits *eagerly*. That is, whenever an event a is added, TruSt performs a local "race-detection" phase (i.e., looks for forward revisits if a is a read, or alternative co positions and backward revisits if a is a write), and, for each of the possible alternatives found, it immediately initiates a recursive exploration. For instance, for W+W+RR and the explorations shown in Fig. 5 and Fig. 6, each of the revisits performed in these explorations can be explored recursively and eagerly, at exactly the time each revisiting event was added in the graph.

Since an executed backward revisit will not be "deleted" from the graph by subsequent backward revisits, the number of events that will never be removed from the graph increases. Since the number of events that can be added in a graph is bounded by the program size, so is the number of recursive calls that can be performed from a given graph. A simple calculation gives us a space complexity bound of $O(n^3)$, where n is the size of the program: the recursion depth is at most n^2 (there are at most n backward revisits, between any pair of which up to n events may have been added) and each recursive call uses $O(n)$ space to store the execution graph.

[3] All writes are co-after the initializing writes.

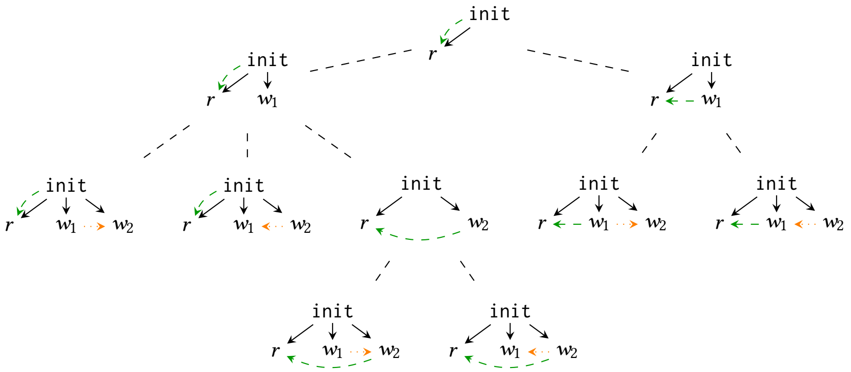

Fig. 8. TruSt: Disjoint explorations for R+W+W

With clever data structures and a more careful calculation, we can bring down TruSt's memory requirements to $O(n)$. The key idea is to store a single execution graph, and to have all recursive calls update the graph *in place* when they are called and to *roll back* their updates when they return. Rolling back forward revisits is easy as they update only one rf or co edge. Further, by executing them in a fixed order, one does not need to remember any information to return to the previous state. Rolling back backward revisits is somewhat more difficult, but can still be achieved by keeping only a constant amount of information per backward revisit. Specifically, as a read may be backward-revisited by a write from a unique configuration (the graph being a maximal extension), to get that configuration it suffices to remove the revisited read-write pair from the graph and to keep adding events maximally until the revisiting write is reached.

Let us now put everything together and see how TruSt verifies a different program, namely the R+W+W program below:

$$a := x \;\Big|\Big|\; x := 1 \;\Big|\Big|\; x := 2 \qquad \text{(R+W+W)}$$

This program has 6 executions under SC, which can be seen at the leaf nodes at the exploration procedure of Fig. 8 (assuming a left-to-right exploration by TruSt).

Since TruSt performs revisits eagerly, when TruSt first encounters $\mathtt{W}(x, 1)$, it can either revisit $\mathtt{R}(x)$ or not; for each of these scenarios, TruSt will initiate a recursive subexploration. (Note that revisiting $\mathtt{R}(x)$ is possible, as the current graph is a maximal extension of the graph resulting if $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$.) Assuming that TruSt first explores the non-revisiting case, it will next add $\mathtt{W}(x, 2)$, in all possible co positions. If $\mathtt{W}(x, 2)$ is added co-maximally, TruSt also has the option of revisiting $\mathtt{R}(x)$, and thus recursively explores that option too. In that case, $\mathtt{W}(x, 1)$ is re-added to the graph, but now it cannot revisit $\mathtt{R}(x)$ because the latter is not maximally added (it has been backward-revisited by a write not in porf-prefix of $\mathtt{W}(x, 1)$). Finally, TruSt explores the second top-level recursive call where $\mathtt{W}(x, 1)$ backward-revisits $\mathtt{R}(x)$. When $\mathtt{W}(x, 2)$ is re-added, it cannot revisit $\mathtt{R}(x)$, since again it is not maximally added before $\mathtt{W}(x, 2)$. TruSt will, however, explore all possible coherence placings for $\mathtt{W}(x, 2)$, thus concluding the verification of this program.

3.6 TruSt: Features and State-of-the-Art

We conclude this section with two observations regarding TruSt and the current state-of-the-art.

First, as mentioned in §1, TruSt can be adapted to operate under the rf equivalence, which can be exponentially coarser than Mazurkiewicz equivalence. Following Kokologiannakis et al. [2019], under such rf partitioning, TruSt does not record co, but rather calculates any *induced* co edges due to the writes that the program reads observe. With this partitioning, TruSt explores only 3 executions for R+W+W, as W$(x, 1)$ and W$(x, 2)$ always remain unordered.

The difficulty when extending TruSt for rf equivalence is that "coherence maximality" no longer applies. As examples like R+W+W demonstrate, there can be multiple writes that are maximal according to the induced co edges, which in turn can create many maximal extensions. We resolve this problem with a tie-braking criterion to select one among these maximal writes: as we show in §4.2, using a deterministic tie-braking criterion (e.g., the $\leq_G$ relation), we can extend TruSt for such a partitioning with minimal changes to the core algorithm.

Second, the structure of TruSt is inherently parallelizable. Even though optimal DPOR algorithms have been parallelized (e.g., [Lång et al. 2020]), such parallelizations concerned the implementation of those algorithms, and not the algorithms themselves, as data sharing was required. TruSt is the first optimal, memory-model-agnostic DPOR that requires absolutely no sharing among different threads, since, as shown in §3.5, different revisits can proceed in a completely disjoint manner.

4 ALGORITHM

In this section, we present the full version of our model-checking algorithm, TruSt. First, in §4.1, we present a variant of TruSt for Mazurkiewicz/Shasha-Snir equivalence, which fully tracks co, similar to the assumption made in §3. Then, in §4.2, we adapt TruSt to work for the coarser reads-from equivalence that avoids tracking co. Finally, in §4.3 and §4.4, we outline the proofs of soundness, completeness, and optimality of our algorithm, and bound its memory consumption, respectively.

4.1 Algorithm Overview

TruSt's algorithm can be seen in Algorithm 1. Given an input program P, Verify verifies P by calling Visit with an execution graph $G_\emptyset$ containing only the initialization event. Subsequently, Visit will enumerate all execution graphs of P in a depth-first manner, and ensure that none of them contains an `error`, denoting a safety violation.

Let us now take a closer look at the Visit function, lying at the heart of the verification procedure. At each step, so long as the current execution graph G remains consistent according to the underlying memory model (Line 4), Visit extends the current graph G by calling $\mathsf{next}_P(G)$.

The function $\mathsf{next}_P(G)$ locates a thread that is not blocked nor finished, adds the corresponding event to $G.\mathsf{E}$ and $\leq_G$ (making it maximal), and returns it via a (Line 5). It does not update $G.\mathtt{rf}$ and $G.\mathtt{co}$. Technically, we assume that there is some total order $<_{\mathsf{next}}$ on events indicating the preference of $\mathsf{next}_P()$ as to which event to add first. We assume that $<_{\mathsf{next}}$ respects the program order (i.e., $\mathsf{po} \subseteq <_{\mathsf{next}}$), and that any read and write events that correspond to the same read-modify-write (RMW) instruction are adjacent in $<_{\mathsf{next}}$. Given a set $\mathsf{avail}(G)$ of available events that could be added to G (i.e., namely, the set of next events of each non-terminated, non-blocked thread of the program), $\mathsf{next}_P(G)$ adds the minimal such event to G w.r.t. $<_{\mathsf{next}}$. In particular, this means that if G contains an event corresponding to the read-exclusive event of a successful RMW instruction without its matching write-exclusive event, then $\mathsf{next}_P(G)$ will return that write-exclusive event (which is anyway immediately po-after it). If there are no available events, we say that the execution graph G is *full*, and $\mathsf{next}_P(G)$ returns $\bot$.

The next action that Visit takes, depends on a itself.

- If a is $\bot$ or `error`, Visit returns (Line 6) or raises an error (Line 8), respectively.
- If a is a read, Visit needs to calculate all possible rf options for the newly added event. To that end, for each write w to the same-location as a (Line 11), it recursively calls Visit on the graph that results if $G.\mathtt{rf}(r)$ is mapped to w. Any inconsistent choices will be subsequently eliminated by the consistency check on Line 4 of the corresponding recursive call.

Algorithm 1 TruSt: Truly Stateless Model-checker (recursive version)

```
1:  procedure Verify($P$)
2:      Visit($P, G_\emptyset$)
3:  procedure Visit($P, G$)
4:      if $\neg\mathsf{consistent}_\mathsf{m}(G)$ then return
5:      switch $a \leftarrow \mathsf{next}_P(G)$ do
6:          case $a = \bot$
7:              return "Visited full execution graph $G$"
8:          case $a \in \mathtt{error}$
9:              exit("error")
10:         case $a \in \mathsf{R}$
11:             for $w \in G.\mathsf{W}_{\mathtt{loc}(a)}$ do
12:                 Visit($P$, SetRF($G, a, w$))
13:         case $a \in \mathsf{W}$
14:             VisitCOs($P, G, a$)
15:             for $r \in G.\mathsf{R}_{\mathtt{loc}(a)}$ such that $\langle r, a\rangle \notin G.\mathsf{porf}$ do
16:                 $\mathit{Deleted} \leftarrow \{e \in G.\mathsf{E} \mid r <_G e \wedge \langle e, a\rangle \notin G.\mathsf{porf}\}$
17:                 if $\forall e \in \mathit{Deleted} \cup \{r\}.$ IsMaximallyAdded($G, e, a$) then
18:                     VisitCOs($P$, SetRF($G|_{G.\mathsf{E}\setminus\mathit{Deleted}}, r, a$), $a$)
19:         case _
20:             Visit($P, G$)
21: procedure VisitCOs($P, G, a$)
22:     for $w_p \in G.\mathsf{W}_{\mathtt{loc}(a)}$ do Visit($P$, SetCO($G, w_p, a$))
23: procedure IsMaximallyAdded($G, e, w$)
24:     $\mathit{Previous} \leftarrow \{w' \in G.\mathsf{E} \mid w' \leq_G e \vee \langle w', w\rangle \in G.\mathsf{porf}\}$
25:     if $\exists r \in \mathit{Previous}$ such that $G.\mathsf{rf}(r) = e$ then return false
26:     $e' \leftarrow$ if $e \in G.\mathsf{R}$ then $G.\mathsf{rf}(e)$ else $e$
27:     return $e' \in \mathit{Previous} \wedge \nexists w' \in \mathit{Previous}.\ \langle e', w'\rangle \in G.\mathsf{co}$
```

- If a is a write, as explained in §3, Visit needs to examine both the case where a does not revisit any of the graph reads, and the case where a revisits some read in G.

To take care of the first case, Visit calls VisitCOs (Line 14). For each possible co-predecessor w_p of a in G, VisitCOs will insert a immediately after w_p in co via SetCO(G, w_p, a), and then call Visit on the resulting graph (Line 22). Formally, SetCO(G, w_p, w) returns a new graph G', which is identical to G except that its $G'.\mathsf{co}$ component is set to:

$$G.\mathsf{co} \cup \{\langle w', w\rangle \mid \langle w', w_p\rangle \in G.\mathsf{co}\} \cup \langle w_p, w\rangle \cup \{\langle w, w'\rangle \mid \langle w_p, w'\rangle \in G.\mathsf{co}\}\,.$$

Dealing with the case where a backward-revisits some read in G is naturally more challenging, as it reflects the essence of TruSt and its maximal extension condition. Visit iterates over all same-location reads as a that are not porf-before a as candidates for a backward revisit (Line 15). (Reads that are porf-before a are excluded because revisiting them would create a porf cycle, which is forbidden by graph well-formedness; see §3.1.) For each candidate read r, Visit calculates the set of events that will be deleted from G if a backward-revisits r (Line 16), and checks if r and every other event e to be deleted was added maximally by calling IsMaximallyAdded (Line 17). If

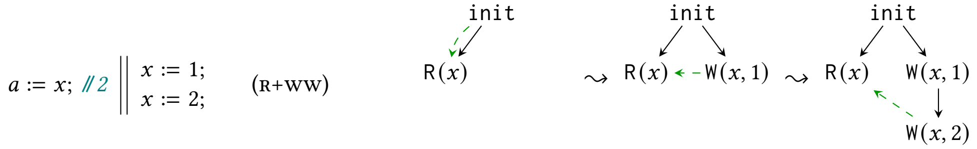

Fig. 9. Revisiting a read multiple times is often necessary

so, Visit appropriately restricts G by removing the deleted events, makes r read from a, and then calls VisitCOs to explore all possible coherence positions for a in the new graph (Line 18), thereby implementing the backward revisiting procedure described in §3.

Accordingly, IsMaximallyAdded(G, e, w) closely follows the definition of the event e being maximally added before w in G (cf. §3.4). First, it calculates the set *Previous* of previous events (i.e., those that were added before e or that are G.porf-before a). Next, it checks whether some other event r that has been backward-revisited by e and, if so, returns false. Then, if e is a write event, it checks that e itself is co-maximal in *Previous*. If e is a read event, it checks that e reads from a co-maximal event in *Previous*. Note that if e is neither a read nor a write (e.g., a fence event), then the maximality check trivially succeeds.

Finally, coming back to the switch-statement of Visit, for all other cases of events (e.g., memory fences), Visit simply initiates a recursive call (Line 19), with no special care taken.

At this point, our presentation of TruSt is complete. However, there are two points worth mentioning regarding Algorithm 1.

The first one is that, when backward-revisiting, Visit sets the coherence placing of a *after* the new graph has been created. In turn, one may wonder: why doesn't it set it before the graph is restricted, so that said placing can be used both for the revisiting and the non-revisiting case? The answer to this question is that, if Visit did that, then it would also have to perform an extra check about the co ordering of a w.r.t. deleted events.

The second point worth mentioning is that Algorithm 1 can backward-revisit a given read multiple times, even though backward revisits are generally preserved. Doing so is frequently necessary to obtain some outcomes. One such example is shown in Fig. 9. For the R+WW example, the execution where $a := x$ reads the annotated value can only be obtained if R(x) is revisited twice: R(x) is not maximally added before W(x, 2) in the subexploration where it is not revisited by W(x, 1) and keeps reading 0, thereby precluding the revisit of R(x) by W(x, 2). More generally, even though maximal extensions forbid backward revisits from deleted events, they do allow backward revisits from porf-related stores.

4.2 TruSt: Adaptation for a Reads-From Equivalence Partitioning

While Algorithm 1 enumerates all consistent execution graphs of a given program, it does so while also tracking full coherence (co) among same-location writes. Indeed, as explained in Section 3.4 and 4.1, the notion of co-maximality is of utmost importance for TruSt when it comes to checking for maximal extensions.

Many recent DPOR approaches, however, avoid tracking full coherence (e.g., [Abdulla et al. 2019; 2018; Chalupa et al. 2017; Kokologiannakis et al. 2019; 2020]). By not doing so, they obtain an exponentially coarser equivalence partitioning, often referred to as *reads-from equivalence partitioning*. While it is unclear whether such a partitioning has an actual impact on the verification time of real-world workloads [Kokologiannakis et al. 2019], it *can* lead to exploring exponentially fewer executions in programs that have unordered, concurrent, same-location writes.

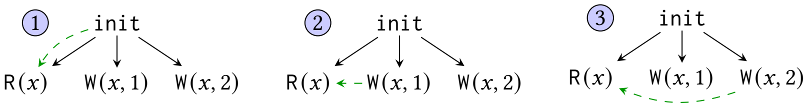

Fig. 10. Executions of R+W+W under a reads-from equivalence partitioning.

Algorithm 2 TruSt: Adaptation for a Reads-From Equivalence Partitioning

```
procedure Visit($P, G$)
    ...
    $Deleted \leftarrow \{e \in G.\mathsf{E} \mid r <_G e \wedge \langle e, a\rangle \notin G.\mathsf{porf}\}$
    $G.\mathsf{co} \leftarrow \mathrm{GetConsCO}(G \setminus Deleted) +\!\!+_{\mathsf{co}} [Deleted \cap \mathsf{W}]; <_G; [Deleted \cap \mathsf{W}]$
        where $a +\!\!+_{\mathsf{co}} b \triangleq \mathsf{a} \cup \mathsf{b} \cup \bigcup_{l \in \mathsf{Loc}} (dom(\mathsf{a}) \cap \mathsf{W}_l) \times (dom(\mathsf{b}) \cap \mathsf{W}_l)$

procedure VisitCOs($P, G, a$)
    Visit($P, G$)
```

As an example of this, consider again the R+W+W program from §3.5 and its executions under a reads-from equivalence partitioning, shown in Fig. 10. Under such a partitioning, the two writes to x remain *unordered*, since their order is not observed by any read of the program. In terms of program verification, a model checker operating under such a partitioning can verify the R+W+W program by enumerating only 3 executions, instead of 6.

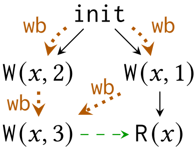

Fig. 11. The wb relation

Generally, the way DPORs avoid ordering unobserved, concurrent writes is by replacing co with a different coherence relation that only *partially orders* same-location writes. In the case of GenMC (and, by extension, TruSt), this relation is called *writes-before* (wb) [Kokologiannakis et al. 2019; Lahav et al. 2015]. To get a taste of how wb works, consider the example in Fig. 11. In the depicted execution graph, the two writes of the first thread are ordered by wb, as W$(x, 2)$ is po-before W$(x, 3)$. In addition, W$(x, 1)$ must be wb-before W$(x, 3)$, as otherwise R(x) would read 1 due to coherence. However, observe that W$(x, 2)$ and W$(x, 1)$ remain unordered: as long as wb is respected for x, these two writes can execute in either order, and this order cannot be observed.

At this point, a question has to be asked: can we unite TruSt's maximal extension idea (that heavily relies on co) with a relation like wb (that generally avoids ordering concurrent writes)?

Fortunately, the answer is yes, but doing so is tricky to achieve. Clearly, one cannot simply replace co with wb in the definition of maximally added events because there may be multiple wb-maximal writes in a consistent execution graph. To maintain optimality, we need a tiebraker between these wb-maximal events. A simple solution is to pick an arbitrary tiebraker (e.g., the write that was inserted last to the graph, i.e., the $<_G$-maximal event among the wb-maximal ones). This solution works but it requires a stronger extensibility property of the underlying memory model: extending a consistent execution graph with a read event that reads from any wb-maximal event should result in a consistent graph. While this property holds of certain simple models, such as release-acquire consistency, it does not hold of other models, including SC. The problem is that while a consistent graph must have its co be some location-total order extending wb, it is not the case that all location-total orders extending wb satisfy the consistency predicate of the model.

In Algorithm 2, we present a better solution, which only assumes that there is a way, given by the function GetConsCO, to calculate a co relation according to which a graph is consistent. A naive implementation of this function is to enumerate all location-total orders extending wb in a systematic fashion, and return the first one that satisfies the consistency predicate of the model. Better implementations can be derived from the more efficient ways of checking consistency of an execution graph that does not contain a co component (e.g., [Abdulla et al. 2019; Biswas et al. 2019; Bui et al. 2021]).

Our adapted algorithm (henceforth TruSt/wb) obtains a suitable co relation by invoking the function GetConsCO on the part of the graph that will remain unchanged by the backward revisit and appending to the end of this order any writes that will be deleted by the revisit following their insertion order. Thus, by construction, all deleted writes will satisfy the maximality condition. The deleted reads will satisfy the maximality condition if they read from the last deleted same-location write that was inserted into the graph before them (if such a write exists) or else from the non-deleted same-location write that was deemed maximal according to the order returned by GetConsCO.

The only other change necessary is the definition of VisitCOs: since TruSt/wb does not track full coherence, VisitCOs simply boils down to a Visit call, thereby showcasing the exponential improvement that TruSt/wb potentially offers.

4.3 TruSt: Soundness, Completeness & Optimality

Assuming that the input program P has executions only of a bounded size, we show that the TruSt algorithm (Algorithm 1) always terminates, and is sound, complete and optimal. Soundness ensures that if $\text{Verify}(P)$ generates G, then G is a consistent full program execution. Completeness ensures that if G is a consistent full execution of P, then $\text{Verify}(P)$ will generate G. Optimality ensures that TruSt generates each execution exactly once and never engages in wasteful explorations. Proofs of these results are given in full in Kokologiannakis et al. [2022]; we proceed with an overview.

Algorithm termination. We first show that once any write w backward-revisits some read r, it cannot be deleted in any subsequent subexploration. Suppose, by contradiction, that w gets deleted by a backward revisit of some previous read r' by a write w'. If it is $r' \leq r$, then r must be in the set of deleted events for the revisit of r', or else w would not get deleted. But r itself cannot be deleted in such a scenario because it has not been added maximally before w': it reads from the deleted event w which was added after it. Otherwise, it must be $r' > r$, but in that case w cannot be deleted because a non-deleted event (r) is reading from it.

Termination of Algorithm 1 follows from the assumption that all executions of P are of bounded size. Since all algorithm steps except for backward revisits increase the graph size, and since writes initiating a backward revisit cannot be removed, there can only be a bounded number of backward revisits, and therefore a bounded number of algorithm steps.

Soundness. TruSt is trivially sound because events are added to the graph following the program semantics, while inconsistent executions are dropped as soon as they are reached.

Completeness. Completeness states that $\text{Verify}(P)$ visits every consistent full graph of P.

Theorem 4.1 (Completeness). *Let G_f be a consistent full execution graph of P. Then $\text{Verify}(P)$ calls $\text{Visit}(P, G'_f)$ for some graph $G'_f \approx G_f$.*

The key idea behind the proof is that, given an execution G reached by the algorithm, we can infer the execution that immediately precedes it in the (unique) production sequence that leads to G. This observation enables us to define a procedure Prev (Algorithm 3) that maps every non-empty

Algorithm 3 Prev: Backward step from G to G_p

```
procedure Prev($P, G$)
    $a \leftarrow \max_{<_{\mathsf{next}}} \{e \in G.\mathsf{E} \mid \nexists e'. \langle e, e' \rangle \in G.\mathsf{porf}\}$    ▹ Get the maximal event
    if $a \in \mathsf{R} \land \langle a, G.\mathsf{rf}(a) \rangle \in <_{\mathsf{next}} \land \nexists b \neq a. \langle G.\mathsf{rf}(a), b \rangle \in G.\mathsf{porf}$ then    ▹ Backward revisit?
        return $\langle$MaxCompletion$(P, G \setminus \{a, w\}, w)$, "$w$ back-revisits $a$"$\rangle$    ▹ Add deleted events
    else
        return $\langle G \setminus \{a\}$, "non-revisit $a$"$\rangle$    ▹ Just remove the event
procedure MaxCompletion($P, G, e$)
    while $a \leftarrow \mathsf{next}_P(G)$ do    ▹ Keep adding events...
        if $a = e$ then return $G \setminus \{e\}$    ▹ ... until we find $e$
        else if $a \in \mathsf{R}$ then $G.\mathsf{rf}[a] \leftarrow \max_{G.\mathsf{co}} \{G.\mathsf{W}_{\mathsf{loc}(a)}\}$    ▹ Read from co-maximal
        else if $a \in \mathsf{W}$ then $G \leftarrow \mathrm{SetCO}(G, \max_{G.\mathsf{co}} \{G.\mathsf{W}_{\mathsf{loc}(a)}\}, a)$    ▹ Make co-maximal
```

consistent execution to its "previous" execution. Prev lets us take a backward step; from G to the unique execution G_p such that Visit(P, G_p) immediately leads to a Visit(P, G) call.

We show that repeatedly taking Prev-steps from G_f will eventually lead to the initial graph: at each point G_f's maximal event is removed or made to read from an earlier event. Next, we show that whenever a graph G is Prev-reachable from a consistent full execution and G_p is a reachable algorithm configuration such that Prev$(G) = \langle G'_p, _ \rangle$ with $G'_p \approx G_p$, then Visit(P, G_p) calls Visit(P, G') for some $G' \approx G$. Then, Theorem 4.1 follows by induction on the sequence of Prev-steps from G_f.

Optimality. Optimality consists of showing two properties: (1) that there are no duplicate explorations, and (2) that there are no fruitless explorations that are doomed to be blocked and can never lead to a full execution.

To establish the former, we first show that for every reachable algorithm configuration G, if G performs an algorithm step t and reaches configuration G', then Prev$(G') = \langle G, t \rangle$. This follows because t will either be adding the maximal event to G (non-revisiting case) or the write read by it (backward-revisit case). In either case, Prev(G') will identify that step and "undo" it.

We can then easily prove that there are no duplicate explorations, in that each configuration G is reached at most once. (Assume by contradiction there are two production sequences that reach the same configuration. However, we have just shown that they must have the exact same last step, and now we have two shorter production sequences reaching the same configuration, which by induction should also agree.)

Theorem 4.2 (no duplicate exploration). *Given a graph G, Verify(P) goes through at most one sequence of nested Visit$(P, _)$ calls before calling Visit(P, G') for some $G' \approx G$.*

To establish the latter property, we need an additional assumption about the memory model concerning the treatment of RMW events. We say that a memory model is *exclusive-write extensible* if, for all consistent graphs G with a po-maximal exclusive read $r \in G.\mathsf{R}^{ex}$ such that there is no exclusive read $r' \in G.\mathsf{R}^{ex}$ with an immediate po-successor (matching) exclusive write and $G.\mathsf{rf}(r') = G.\mathsf{rf}(r)$, adding its corresponding exclusive write event w yields a consistent graph (for some choice of co).

Then, we can show that if a reachable algorithm configuration is fruitless, then it is immediately blocked. In fact, the only way a fruitless configuration could arise is by adding the read-exclusive part of an RMW event reading from a write already read by another RMW. The immediate next step will add its write-exclusive part, thereby making the graph inconsistent.

Algorithm 4 TruSt: Iterative version with linear memory requirements

```
 1: procedure VISIT($P, G$)
 2:     while true do
 3:         $a \leftarrow$ if $\mathsf{consistent_m}(G)$ then $\mathsf{next}_P(G)$ else $\bot$
 4:         if $a \in \mathtt{error}$ then exit(“error”)
 5:         else if $a \in \mathsf{R}$ then $G.\mathtt{rf}[a] \leftarrow \max_{<_G} G.\mathsf{W}_{\mathtt{loc}(a)}$
 6:         else if $a \in \mathsf{W}$ then $B[a] \leftarrow a$
 7:         else if $a = \bot$ then
 8:             while true do
 9:                 $a \leftarrow \max_{<_G} G.\mathsf{E}$
10:                 if $a = \bot$ then return
11:                 else if $a \in \mathsf{R} \wedge \exists w \in G.\mathsf{W}_{\mathtt{loc}(a)}.\, w <_G G.\mathtt{rf}(a)$ then
12:                     $G.\mathtt{rf}[a] \leftarrow \max_{<_G}\{w \in G.\mathsf{W}_{\mathtt{loc}(a)} \mid w <_G G.\mathtt{rf}(a)\}$
13:                     break
14:                 else if $a \in \mathsf{W} \wedge \exists r \in G.\mathsf{R}_{\mathtt{loc}(a)}.\, r <_G B[a] \wedge \langle r, a\rangle \notin G.\mathsf{po}\mathtt{rf}$ then
15:                     $B[a] \leftarrow \max_{<_G}\{r \in G.\mathsf{R}_{\mathtt{loc}(a)} \mid r <_G B[a] \wedge \langle r, a\rangle \notin G.\mathsf{po}\mathtt{rf}\}$
16:                     $Deleted \leftarrow \{e \in G.\mathsf{E} \mid B[a] <_G e \wedge \langle e, a\rangle \notin G.\mathsf{po}\mathtt{rf}\}$
17:                     $G.\mathtt{co} \leftarrow \mathrm{GetConsCO}(G \setminus Deleted) +\!\!+_{\mathtt{co}} [Deleted \cap \mathsf{W}]; <_G; [Deleted \cap \mathsf{W}]$
18:                     if $\forall e \in Deleted \cup \{B[a]\}.$ ISMAXIMALLYADDED($G, e, a$) then
19:                         $G.\mathtt{rf}[B[a]] \leftarrow a$
20:                         $G \leftarrow G \setminus Deleted$
21:                         break
22:                 else switch PREV($G$) do
23:                     case $\langle \_,$ “non-revisit $e$”$\rangle$
24:                         $G \leftarrow G \setminus \{e\}$
25:                     case $\langle G_p,$ “$e$ back-revisits $a$”$\rangle$
26:                         $\leq' \leftarrow \{\langle x, y\rangle \in G.\mathsf{E} \times G.\mathsf{E} \mid x \leq_G y \wedge y <_G a\}$
27:                         while $d \leftarrow \min_{<_{\mathsf{next}}} (G_p.\mathsf{E} \setminus dom(\leq'))$ do
28:                             $B \leftarrow \{b \in G.\mathsf{E} \cap G_p.\mathsf{E} \setminus dom(\leq') \mid b \leq_G G_p.\mathtt{rf}(d)\}$
29:                             $\leq' \leftarrow \leq' +\!\!+ \{\langle d, d\rangle\} +\!\!+ \{\langle x, y\rangle \in B \times B \mid x \leq_G y\}$
30:                         $G \leftarrow G_p$ ; $\leq_G \leftarrow \leq'$
```

4.4 TruSt: Iterative Version with Linear Memory Consumption

Finally, we present an iterative version of the TruSt algorithm that has linear memory consumption. For simplicity, in Algorithm 4, we show the version that does not record co and works for rf-equivalence. The algorithm clearly has linear space complexity, as it keeps only one copy of the execution graph G together with an auxiliary array B for tracking backward revisits, and does not call itself recursively.

Algorithm 4 operates in an iterative fashion in one of two modes: (1) the *forward mode*, which keeps adding events to the graph while possible; and (2) the *backtracking mode*, which changes rf edges of graph when alternative exploration options are possible, removes events (e.g., to perform a backward revisit or when all revisit options of an event have been explored), and/or restores events that were removed by a backward revisit that needs to be undone.

The forward mode corresponds to the outer **while** loop of Algorithm 4 and is quite straightforward. As long as the graph is consistent, the graph is extended with the next available event a (Line 3). If that event signifies an error, verification fails with an error message (Line 4). If a is a read, its rf is set to the maximal write of the same location according to insertion order. If a is a write, we initialize its index in the B array. Otherwise, if the execution is complete (or inconsistent), we enter into the backtracking mode (Line 8).

The backtracking mode corresponds to the inner **while** loop and is a bit more subtle. It starts by selecting the maximal event a from G (Line 9). If no such event exists (i.e., the graph contains only the initialization event), backtracking is complete, and so verification finishes (Line 10). Now if a exists and is a read event, we have to examine whether a has any remaining forward revisiting options that were not considered. If there are further possible writes where a can read from, earlier in insertion order than the write a is currently reading (Line 11), then we set a to read from the maximal such write (Line 12), and go back into the forward mode (Line 13).

Similarly, if a is a write event, we have to examine whether there are any (further) reads that need to be backward-revisited by a. If there are such reads (Line 14), we select the latest according to insertion order, and store it in $B[a]$ (Line 15). Then, if the selected read satisfies the maximal extension condition (Line 18), we update its rf to read from a (Line 19), restrict the graph (Line 20), and go back into the forward mode (Line 21).

Finally, if a does not have any remaining revisit options, we call $\textsc{Prev}(G)$ (Algorithm 3) that returns the previous execution step (Line 22). If that is not a backward-revisit step, we simply remove the maximal event from G (Line 23). If, however, it was a backward revisit from a graph G_p (Line 25), we need to do some more work to reconstruct the correct sequence of events in G_p. For this, we follow the order of events in G for events prior to a, and the insertion order for the deleted events. Whenever a deleted event d reads from a later (in insertion order) event of G, this means that d had been backward-revisited; thus, we also add all prior (not yet added) events of G immediately after d: the operation $\mathsf{r_a} +\!\!+ \mathsf{r_b}$ returns $\mathsf{r_a} \cup \mathsf{r_b} \cup \mathit{dom}(\mathsf{r_a}) \times \mathit{dom}(\mathsf{r_b})$.

5 IMPLEMENTATION

In Section 1 and 3, we stressed that memory-model parametricity is one of the key features of TruSt. In order to have an implementation that is also parametric in the choice of the memory model, we implemented TruSt on top of GenMC [Kokologiannakis et al. 2019]. GenMC's implementation is open-source (https://github.com/mpi-sws/genmc), and comes with built-in support for weak memory models such as RC11 [Lahav et al. 2017].

That said, and even though GenMC's architecture is generally modular and does allow for extensions, we had to modify GenMC's existing implementation significantly to integrate the changes necessary for TruSt and also to accommodate for TruSt's parallelization. The latter part was arguably the most challenging in technical terms.

As far as the sequential implementation of TruSt is concerned, our implementation mostly follows the recursive version of the TruSt algorithm, but performs forward revisits in place, and only copies the graph for backward revisits. Although this means that our implementation consumes quadratic space, we believe that it is faster than the purely iterative version of TruSt that needs to rebuild the execution graph after each backward revisit by re-interpreting the program. Moreover, as our experiments confirm (§6.1), the memory consumption of our implementation is never an issue.

As far as the parallel implementation of TruSt is concerned, we faced two major challenges. The first challenge was that GenMC uses several LLVM intrinsics that are not thread-safe. Thus, to preclude concurrency bugs due to the internal LLVM libraries on which GenMC relies, we redesigned a significant portion of GenMC's infrastructure to reduce its reliance on LLVM intrinsics, so that different threads can communicate in a thread-safe manner.

Another issue we had to deal with was the design of the communication among different threads. For that, we opted for the following simple solution: since TruSt copies the current execution graph whenever it initiates a recursive backward-revisit exploration, in a multicore setting, it can simply pass the graph copy to another worker thread, so that the two explorations can proceed in parallel. Granted, such a design relies on there being enough backward revisits in a given program, but we nevertheless found that it provides good scalability in practice. Even though we also tried providing each thread with its own work-stealing queue, we found that this approach did not yield any benefits over our current implementation.

Putting everything together, we obtained an implementation of TruSt that has polynomial memory requirements and takes full advantage of the underlying machine's parallelism (§6.2). As we show in §6.1, TruSt's performance is on par with state-of-the-art DPOR implementations.

6 EVALUATION

Our evaluation revolves around validating the claims of §1. Concretely, we set out to show the following two points:

(1) TruSt provides the best of both worlds: optimality and polynomial memory consumption.
(2) TruSt is inherently parallelizable: it scales up to a large number of cores because different explorations share no state.

To demonstrate the first point (§6.1), we performed an evaluation comprising two parts. In the first part, we compared TruSt against two other stateless model checkers, namely GenMC and Nidhugg [Abdulla et al. 2015; 2014]. We chose GenMC because TruSt is built on top of it (and also because its DPOR algorithm is optimal), and Nidhugg because it provides both an optimal and a nonoptimal (sleepset-based) DPOR algorithm, thus nicely highlighting the tradeoff between time and memory that DPOR algorithms have to pay. In the second part, we evaluated TruSt against GenMC on a set of realistic, weak-memory benchmarks. As Nidhugg does not support a memory model similar to GenMC's RC11[4], and memory consumption directly depends on the number of executions allowed by the model, we had to exclude it from this comparison. As we show in §6.1, TruSt is always exponentially faster than nonoptimal DPOR implementations, and exponentially "lighter" (in terms of memory consumption) than optimal DPOR implementations. Even though TruSt can be polynomially slower than an optimal DPOR, we observe that the overhead that TruSt faces is insignificant. By contrast, in cases where optimal DPORs consume an exponential amount of memory, TruSt can be exponentially faster, since its computations do not become memory-bound. To keep the comparison fair, in both parts, we run TruSt with only one worker thread because the other tools do not support the same degree of parallelism.

To demonstrate the second point (§6.2), we evaluated TruSt's parallel implementation with an increasing number of threads on a number of larger benchmarks. As we show in §6.2, our prototype implementation achieves an almost linear speedup when scaling up to the number of physical cores available, thus yielding significant performance improvements.

Experimental Setup. We conducted all experiments on a Dell PowerEdge M620 blade system, running a custom Debian-based distribution, with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz), and 256GB of RAM. We used LLVM 7.0.1 for Nidhugg, GenMC and TruSt. Unless explicitly noted otherwise, all reported times are in seconds. We set a timeout limit of 24 hours and a memory limit of 500MB.

6.1 TruSt vs State-of-the-Art

Let us begin with some synthetic benchmarks that highlight the differences between the different DPOR algorithms (cf. Table 1). The first benchmark in Table 1, `lastzero` from Abdulla et al.

[4] Nidhugg's support for POWER/ARMv7 is based on a different (nonoptimal) algorithm, and spans over a limited subset of these models.

Table 1. Synthetic benchmarks

	Nidhugg/source			Nidhugg/optimal			GenMC			TruSt		
	Executions	*Mem.*	*Time*	*Executions*	*Mem.*	*Time*	*Executions*	*Mem.*	*Time*	*Executions*	*Mem.*	*Time*
lastzero(10)	20 195	49	6	3328	49	1	3328	46	0	3328	47	0
lastzero(15)	4 799 353	51	2137	147 456	494	102	147 456	46	9	147 456	47	11
lastzero(20)	OOM	OOM	OOM	OOM	OOM	OOM	6 029 312	46	455	6 029 312	47	557
exp-mem(7)	10 080	48	2	10 080	81	2	10 080	182	1	10 080	46	1
exp-mem(8)	80 640	49	18	OOM	OOM	OOM	OOM	OOM	OOM	80 640	46	10
exp-mem(9)	725 760	49	176	OOM	OOM	OOM	OOM	OOM	OOM	725 760	46	103
exp-mem2(4)	21 386	48	4	20 736	48	4	20 736	55	1	20 736	46	1
exp-mem2(5)	746 378	48	182	705 600	48	177	705 600	326	58	705 600	46	52
exp-mem2(6)	36 044 140	251	11 194	33 177 600	240	10 884	OOM	OOM	OOM	33 177 600	46	2757

[2017], is a prime example of (1) the memory/time tradeoff that DPOR algorithms have to face, and (2) the different backtracking strategies that DPOR algorithms employ. As can be seen in Table 1, Nidhugg/optimal is exponentially faster than Nidhugg/source for `lastzero(10)` and `lastzero(15)`, as it explores exponentially fewer executions than Nidhugg/source. That speed however, comes at a price: Nidhugg/optimal also consumes an exponential amount of memory, which makes it exceed the memory limit when the number of threads is increased to 20.

That said, perhaps surprisingly, Nidhugg/source also exceeds the memory limit for `lastzero(20)`. This, however, is due to a different reason: Nidhugg/source's backtracking strategy (§ 2.1). Indeed, because Nidhugg/source explores an exponential number of sleepset-blocked executions for `lastzero`, it populates the trace's sleep and backtrack sets with an exponential number of transitions to fire, thus consuming an exponential amount of memory. GenMC and TruSt, on the other hand, both consume much less memory: GenMC due to its graph-based backtracking (see § 3.2) and the small amount of backward revisits it performs, and TruSt due to the memory guarantees it provides by design. (Note that, since TruSt also performs extra checks for maximal extensions compared to GenMC, it also has a small overhead.)

As the next two benchmarks show, however, it is not at all hard for GenMC to exceed the memory limit as well. The `exp-mem` benchmark (adapted from Abdulla et al. [2017]) is another example where optimal DPORs explore an exponential amount of memory. Here, both GenMC and Nidhugg/optimal quickly exceed the memory limit, while TruSt verifies all variants of this program with essentially the same memory consumption. Analogously, for `exp-mem2` (a variant of `exp-mem` with no RMW operations), both Nidhugg/optimal and GenMC consume an exponential amount of memory for 6 threads and above, while TruSt maintains the same memory consumption, as expected. (Nidhugg/source consumes a large amount of memory due to its backtracking strategy, as already explained for `lastzero`.) The reason why Nidhugg/optimal consumes less memory than GenMC in general, is due to the way it stores transition sequences in its backtrack sets: since Nidhugg/optimal only operates under SC, it can very compactly represent such transition sequences, thus leading to lower memory consumption in general. As a final note before moving on to the next table, observe that TruSt outperforms GenMC for `exp-mem2`, as TruSt's exploration does not become memory-bound, in contrast to GenMC.

Moving on to the second part of this evaluation (cf. Table 2), we observe that the same high-level claims and trends we made for Table 1 also extend to realistic benchmarks. As can be seen in Table 2, as the number of threads increases, GenMC quickly exceeds the memory limit, while TruSt's memory consumption basically remains constant. In addition, even though GenMC is usually slightly faster than TruSt, whenever an exploration becomes memory-bound (e.g., `seqlock`), TruSt outperforms GenMC in terms of both memory and time.

Table 2. Weak memory benchmarks

	GenMC			TruSt		
	Executions	*Mem.*	*Time*	*Executions*	*Mem.*	*Time*
lamport(3)	6690	47	0	6690	46	0
lamport(4)	12 163 630	299	1122	12 163 630	46	1007
mcs_spinlock(4)	15 264	78	7	15 264	47	8
mcs_spinlock(5)	OOM	OOM	OOM	964 320	48	617
ttaslock(3)	162	47	0	162	48	0
ttaslock(4)	20 760	52	2	20 760	47	3
ttaslock(5)	OOM	OOM	OOM	14 457 720	48	2545
mpmc_queue(3)	143	46	0	143	47	0
mpmc_queue(4)	31 880	54	238	31 880	47	251
mpmc_queue(5)	OOM	OOM	OOM	1 270 584	48	37 835
seqlock-atomic(5)	1500	99	18	1500	47	14
seqlock-atomic(6)	OOM	OOM	OOM	16 200	47	373
seqlock-atomic(7)	OOM	OOM	OOM	185 220	47	9138
mutex_musl(3)	361 296	59	39	361 296	48	44
mutex_musl(4)	OOM	OOM	OOM	🕒	🕒	🕒

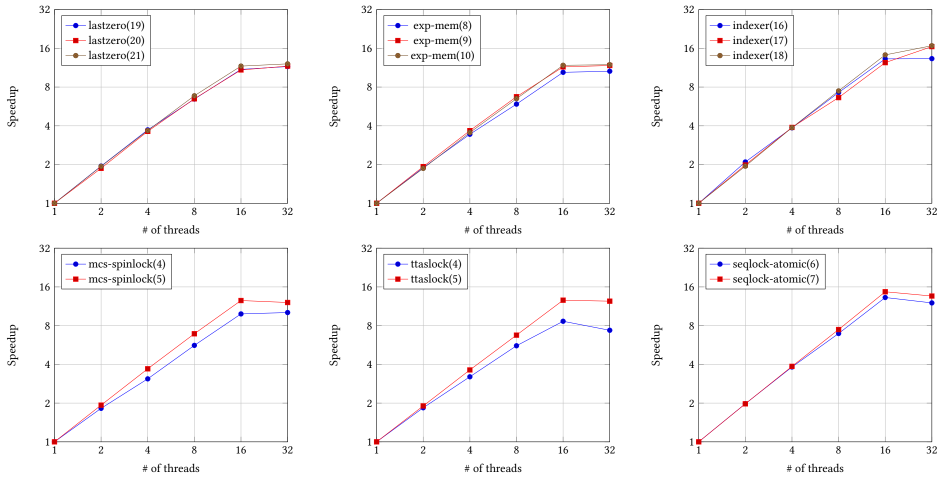

Fig. 12. Scalability of TruSt on an architecture with 16 physical cores (32 logical cores)

Finally, let us conclude with an observation. In order to stress the importance of polynomial memory consumption, we let the `mutex_musl` benchmark run on our server without a memory limit, so that we measure the exact memory consumption of the test case, and see how GenMC is affected by it. (We selected this particular example because it is super-exponential in terms of executions and thus occupies a lot of space.) To our surprise, GenMC consumed all 256 GB(!) of our server's RAM, effectively thrashing the system. TruSt, on the other hand, even though it did not terminate within the time limit, did not consume more than 50 MB until it timed out.

6.2 Parallelizing TruSt

Let us now see how TruSt scales when we increase the number of threads on our multicore architecture of 16 physical cores (32 logical cores with hyperthreading). Figure 12 plots the speedup obtained by running TruSt over single-threaded performance (y-axis) against the number of worker threads employed (x-axis). Both axes are in logarithmic scale, so that perfect scaling would correspond to the diagonal line.

As can be seen, TruSt achieves an almost linear speedup when scaling up to 16 threads, and then its performance flattens out and even deteriorates as we add more worker threads than physical cores on the machine. The speedup obtained by using up to 16 worker threads dramatically decreases the running time for some very intensive benchmarks. As an example, consider the `seqlock-atomic` benchmark: while the sequential version of TruSt needs more than 2.5 hours to terminate for this benchmark, with 16 cores it terminates in 10 minutes.

There are three additional takeaways from Fig. 12. First, TruSt's speedup is almost, but not exactly, linear, up to 16 cores. This is expected and in line with most results of parallel algorithms. Indeed, even though the design of TruSt allows for explorations to proceed completely in parallel, there is no guarantee that all different subexplorations will have the same "depth". Specifically, each thread necessarily has some small "idling" period where it tries to pick up work from other threads, thus limiting the scalability of TruSt.

Second, scalability flattens at 16 worker threads even though our machine does support hyperthreading and thus is deemed to have 32 logical cores. Again, this is expected because our computations are CPU/memory-bound (as opposed to I/O-bound) and thus hyperthreading does not succeed in running more tasks in parallel.

Third, TruSt scales better when either the state space of a benchmark or the cost per execution becomes larger. For instance, TruSt scales generally better for the last four benchmarks of Fig. 12 than for the first two of the same figure: this is because the per-execution cost of the weak-memory benchmarks is larger, which in turn means that the different threads have more work to do before trying to pick up their next tasks. In addition, observe that TruSt scales better for each benchmark as we increase the parameter controlling its state space. Again, this is expected as a larger state space entails more executions, which in turn implies that each of TruSt's worker threads will have more work to do.

7 RELATED WORK

After seminal works like VERISOFT [Godefroid 1997; 2005] and CHESS [Musuvathi et al. 2008] paved the way for stateless model checking, there has been a large body of work on SMC and DPOR [Flanagan et al. 2005]. A major breakthrough in this line of work was made by Abdulla et al. [2014], who provided the first optimal DPOR algorithm for Mazurkiewicz trace equivalence for sequential consistency. This algorithm, as described in §2.3, avoids blocked explorations at the cost of exponential memory consumption. We can broadly classify the more recent works in this area into two main categories depending on their primary focus.

First, many techniques aim to combat the state-space explosion problem by introducing coarser equivalence partitionings [Abdulla et al. 2019; Agarwal et al. 2021; Albert et al. 2017; 2018; Aronis et al. 2018; Chalupa et al. 2017; Chatterjee et al. 2019]. Among these, only NIDHUGG [Abdulla et al. 2019; Aronis et al. 2018] is optimal w.r.t. to its equivalence partitioning(s), although, as demonstrated in §6.1, can suffer from exponential memory consumption.

Second, other techniques focus on extending DPOR to weak memory models either by targeting a specific weak memory model [Abdulla et al. 2015; 2016; 2018; Norris et al. 2013] or by being parametric with respect to an axiomatically-defined memory model [Kokologiannakis et al. 2019;

2020]. In addition, a number of these works can operate under the coarser reads-from equivalence partitioning, including Abdulla et al. [2018] and Kokologiannakis et al. [2019, 2020], which do so while maintaining optimality. TruSt builds on the foundations laid by GenMC [Kokologiannakis et al. 2019] in order to achieve memory-model parametricity; however, it adapts the "extensibility" requirement on the underlying memory model to a more precise "maximal extensibility" requirement (see §3.1)

In contrast to TruSt, however, none of the techniques above manages to combine (a) being parametric w.r.t. the memory model, (b) operating under reads-from equivalence, (c) being optimal, and (d) maintaining polynomial memory consumption. To some degree, the combination between (c) and (d) has been explored by Nguyen et al. [2018], with Quasi-optimal DPOR. Quasi-optimal DPOR is able to approximate an optimal DPOR with a user-provided constant k. As the value of k increases, memory consumption also increases in an exponential manner. Although Quasi-optimal DPOR theoretically achieves optimality only with $k = \infty$, it has been shown to practically be optimal, for small values of k.

Finally, it is worth wondering whether the idea of maximal extensions could be used to achieve optimality for an algorithm like that of RCMC [Kokologiannakis et al. 2017], which is not strictly a DPOR algorithm. Unfortunately, it turns out that maximal extensions do not suffice; to see this, consider the program below:

$$\begin{array}{l} r_1 := x; \\ y := 1; \end{array} \left\| \; r_2 := y; \;\right\| \; x := 1; \qquad \text{(WR+R+W)}$$

In contrast to TruSt, when backward-revisiting, RCMC deletes only the porf-suffix of the revisited events, keeping the rest of the graph intact. Accordingly, when integrating maximal extensions in RCMC, RCMC is able to revisit the read of x both when $r_2 = 0$ and when $r_2 = 1$, thus leading to an execution where $r_1 = 1 \land r_2 = 0$ twice. With an additional constraint, it may well be possible to obtain optimality; we leave that to future work.

8 CONCLUSION & FUTURE WORK

We presented TruSt, an optimal, memory-model-parametric stateless model checking algorithm that reconciles SMC with DPOR. TruSt provides the *first* optimal DPOR framework that maintains polynomial memory consumption. In contrast to previous works, TruSt only reverses a race between two memory accesses if the events to be removed from the current execution graph form a maximal extension, i.e., they have been added in a co-maximal way.

In the future, we plan pursue several different research directions. First, we plan to relax the porf-acyclicity assumption of TruSt so that it can also account for hardware memory models like POWER [Alglave et al. 2014] and ARMv8/RISC-V [Pulte et al. 2018]. Second, we plan to fine-tune and optimize TruSt's parallel implementation, by designing a more elaborate message-passing scheme that will allow for even greater scalability in larger benchmarks. Finally, we plan to leverage TruSt's low memory requirements and use it to verify critical components of large software systems (e.g., data-structures of the Linux kernel).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES

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 2015*. Vol. 9035. LNCS. Berlin, Heidelberg: Springer, pp. 353–367. DOI: https://doi.org/10.1007/978-3-662-46681-0_28.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas (2014). "Optimal dynamic partial order reduction." In: *POPL 2014*. New York, NY, USA: ACM, pp. 373–384. DOI: https://doi.org/10.1145/2535838.2535845.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas (Sept. 2017). "Source sets: A foundation for optimal dynamic partial order reduction." In: *J. ACM* 64.4, 25:1–25:49. DOI: https://doi.org/10.1145/3073408.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas (Oct. 10, 2019). "Optimal stateless model checking for reads-from equivalence under sequential consistency." In: *Proc. ACM Program. Lang.* 3 (OOPSLA), 150:1–150:29. DOI: https://doi.org/10.1145/3360576.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson (2016). "Stateless model checking for POWER." In: *CAV 2016*. Vol. 9780. LNCS. Berlin, Heidelberg: Springer, pp. 134–156. DOI: https://doi.org/10.1007/978-3-319-41540-6_8.

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

Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman (July 2021). "Stateless Model Checking Under a Reads-Value-From Equivalence." In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, pp. 341–366. DOI: https://doi.org/10.1007/978-3-030-81685-8_16.

Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey (2017). "Context-sensitive dynamic partial order reduction." In: *CAV 2017*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, pp. 526–543. DOI: 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: *CAV 2018*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, pp. 392–410. DOI: https://doi.org/10.1007/978-3-319-96142-2_24.

Jade Alglave, Luc Maranget, and Michael Tautschnig (July 2014). "Herding cats: Modelling, simulation, testing, and data mining for weak memory." In: *ACM Trans. Program. Lang. Syst.* 36.2, 7:1–7:74. DOI: https://doi.org/10.1145/2627752.

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas (2018). "Optimal dynamic partial order reduction with observers." In: *TACAS 2018*. Vol. 10806. LNCS. Springer, pp. 229–248. DOI: https://doi.org/10.1007/978-3-319-89963-3_14.

Ranadeep Biswas and Constantin Enea (Oct. 2019). "On the Complexity of Checking Transactional Consistency." In: *Proc. ACM Program. Lang.* 3.OOPSLA. DOI: https://doi.org/10.1145/3360591.

Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman (Oct. 2021). "The Reads-from Equivalence for the TSO and PSO Memory Models." In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: https://doi.org/10.1145/3485541.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya (Dec. 2017). "Data-centric dynamic partial order reduction." In: *Proc. ACM Program. Lang.* 2.POPL, 31:1–31:30. DOI: https://doi.org/10.1145/3158119.

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

Cormac Flanagan and Patrice Godefroid (2005). "Dynamic partial-order reduction for model checking software." In: *POPL 2005*. New York, NY, USA: ACM, pp. 110–121. DOI: https://doi.org/10.1145/1040305.1040315.

Patrice Godefroid (1997). "Model checking for programming languages using VeriSoft." In: *POPL 1997*. Paris, France: ACM, pp. 174–186. DOI: https://doi.org/10.1145/263699.263717.

Patrice Godefroid (Mar. 2005). "Software Model Checking: The VeriSoft Approach." In: *Form. Meth. Syst. Des.* 26.2, pp. 77–101. DOI: https://doi.org/10.1007/s10703-005-1489-x.

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis (Dec. 2017). "Effective stateless model checking for C/C++ concurrency." In: *Proc. ACM Program. Lang.* 2.POPL, 17:1–17:32. DOI: https://doi.org/10.1145/3158105.

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis (Jan. 2022). "Truly Stateless, Optimal Dynamic Partial Order Reduction (supplementary material)." In: URL: https://plv.mpi-sws.org/genmc.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis (2019). "Model checking for weakly consistent libraries." In: *PLDI 2019*. New York, NY, USA: ACM. DOI: https://doi.org/10.1145/3314221.3314609.

Michalis Kokologiannakis and Viktor Vafeiadis (2020). "HMC: Model checking for hardware memory models." In: *ASPLOS 2020*. ASPLOS '20. Lausanne, Switzerland: ACM, pp. 1157–1171. DOI: https://doi.org/10.1145/3373376.3378480.

Ori Lahav and Viktor Vafeiadis (2015). "Owicki-Gries Reasoning for Weak Memory Models." In: *ICALP 2015*. Vol. 9135. LNCS. Springer, pp. 311–323. DOI: https://doi.org/10.1007/978-3-662-47666-6_25.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer (2017). "Repairing sequential consistency in C/C++11." In: *PLDI 2017*. Barcelona, Spain: ACM, pp. 618–632. DOI: https://doi.org/10.1145/3062341.3062352.

Leslie Lamport (Sept. 1979). "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs." In: *IEEE Trans. Computers* 28.9, pp. 690–691. DOI: https://doi.org/10.1109/TC.1979.1675439.

Magnus Lång and Konstantinos Sagonas (2020). "Parallel Graph-Based Stateless Model Checking." In: *ATVA 2020*. Ed. by Dang Van Hung and Oleg Sokolsky. Cham: Springer International Publishing, pp. 377–393. DOI: https://doi.org/10.1007/978-3-030-59152-6_21.

Antoni Mazurkiewicz (1987). "Trace Theory." In: *PNAROMC 1987*. Vol. 255. LNCS. Berlin, Heidelberg: Springer, pp. 279–324. DOI: https://doi.org/10.1007/3-540-17906-2_30.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu (2008). "Finding and reproducing Heisenbugs in concurrent programs." In: *OSDI 2008*. USENIX Association, pp. 267–280. URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf (visited on Nov. 16, 2020).

Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci (2018). "Quasi-optimal partial order reduction." In: *CAV 2018*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. LNCS. Springer, pp. 354–371. DOI: https://doi.org/10.1007/978-3-319-96142-2_22.

Brian Norris and Brian Demsky (2013). "CDSChecker: Checking concurrent data structures written with C/C++ atomics." In: *OOPSLA 2013*. ACM, pp. 131–150. DOI: https://doi.org/10.1145/2509136.2509514.

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell (2018). "Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8." In: *Proc. ACM Program. Lang.* 2.POPL, 19:1–19:29. DOI: https://doi.org/10.1145/3158107.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen (July 2010). "X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors." In: *Commun. ACM* 53.7, pp. 89–97. DOI: https://doi.org/10.1145/1785414.1785443.

Dennis Shasha and Marc Snir (Apr. 1988). "Efficient and correct execution of parallel programs that share memory." In: *ACM Trans. Program. Lang. Syst.* 10.2, pp. 282–312. DOI: https://doi.org/10.1145/42190.42277.

SPARC International Inc. (1994). *The SPARC architecture manual (version 9)*. Prentice-Hall.

Naling Zhang, Markus Kusano, and Chao Wang (2015). "Dynamic partial order reduction for relaxed memory models." In: *PLDI 2015*. New York, NY, USA: ACM, pp. 250–259. DOI: https://doi.org/10.1145/2737924.2737956.