Reconciling Preemption Bounding with DPOR

Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis

MPI-SWS, Kaiserslautern and Saarbrücken, Germany
{imarmanis,michalis,viktor}@mpi-sws.org

Abstract. There are two major techniques for scaling up stateless model checking: *dynamic partial order reduction* (DPOR), which only explores executions that differ in the ordering of racy accesses, and *preemption bounding*, which only explores executions containing up to k preemptions (preemptive context-switches).
Combining these two techniques is challenging because DPOR-equivalent executions often contain a different number of preemptions, making it incorrect to cut explorations that exceed the preemption bound. To restore completeness, prior work has weakened the DPOR algorithm, which often results in the exploration of many redundant executions.
We propose an alternative approach. Starting from an optimal DPOR algorithm, we achieve completeness by allowing some slack on the preemption-bound of the explored executions. We prove that the required slack does not exceed the number of threads of the program (minus two), and that this upper limit is tight.

1 Introduction

Stateless model checking (SMC) [12] is an effective bug-finding technique for concurrent programs that systematically explores all interleavings of the given input program. As such, it suffers from the state-space explosion problem: the number of possible interleavings of a program grows rapidly with the program size. There are two main approaches to attack this problem in the literature.

Dynamic partial order reduction (DPOR) [11] is based on the idea that permutations of *independent* instructions in an interleaving lead to the same state. DPOR deems such interleavings equivalent and strives to explore only one representative interleaving from each equivalence class.

Preemption bounding (PB, a.k.a. context bounding) [26] is based on the idea that concurrency bugs in practice can be exposed with a small number of preemptions [25]. Leveraging this insight, PB only explores the interleavings that arise with at most k preemptions (for some fixed k), thereby guaranteeing a partial coverage of the state space.

Combining the two approaches is non-trivial. Simply modifying a DPOR algorithm to discard any explored executions that exceed the desired bound k is not complete, as executions with $\leq k$ preemptions are missed. To restore completeness, Coons et al. [10] weaken DPOR by adding extra backtracking points, but such an

approach negates any optimality properties of the underlying DPOR algorithm, and can lead to the (redundant) exploration of multiple equivalent interleavings.

In this paper, we propose a different approach. We adapt a state-of-the-art optimal DPOR algorithm with polynomial memory requirements called TruSt [16] to support preemption-bounded search.

We first observe that the preemption-bound definition of Coons et al. [10] is overly pessimistic for incomplete executions (i.e., executions where at least one thread is enabled) in that an incomplete execution can often be extended to a complete one with a smaller preemption-bound. Updating the definition to be more optimistic, however, does not fully resolve the issue: an intermediate execution that exceeds the bound might still be needed in order to reveal a conflicting instruction that leads to the exploration of the desired execution.

Our solution is to allow the exploration of executions exceeding the bound, as long as they only exceed it by a small amount, which we call *slack*. For programs with $N \geq 2$ threads, we show that a slack value of $N - 2$ suffices to maintain completeness (up to the provided bound). Unlike Coons et al. [10], our approach is optimal in the sense that it does not explore equivalent executions more than once. Although it may explore executions with larger bound than the desired one, we argue that these executions are useful, because they can still reveal bugs.

We have implemented our bounding approach in GenMC [19], a state-of-the-art open-source stateless model checker. We show that for small preemption bounds (and despite the slack), bounded search can perform significantly faster than full search. Moreover, we experimentally confirm the literature observation that small bounds suffice to expose most concurrency bugs. We therefore argue that our combination of preemption bounding and DPOR is useful as a practical testing approach, which also provides certain coverage guarantees.

2 Background

In this section, we recall the basic DPOR approach and how prior work has tried to incorporate preemption-bounded search into it. Subsequently, we review the TruSt algorithm [16], which we later build upon to obtain our results.

2.1 The Basics of Dynamic Partial Order Reduction

DPOR starts by exploring one thread interleaving. In the process, it detects conflicting transitions, i.e., instructions that, if executed in the opposite order, will alter the state of the system. At each state, when an earlier transition t is in conflict with a possible transition t' that can be taken by another thread in this state, DPOR considers the execution where t' is fired before t. To accomplish this, DPOR adds the transition t' to the *backtrack set* of the state immediately before t was fired, to be explored later.

We illustrate DPOR by running it on the following example (Fig. 1).

$$\begin{array}{l|\!|l} (r_x)\ a := x & (w_1)\ y := 1 \\ (r_y)\ b := y & (w_2)\ y := 2 \end{array} \qquad \text{(RR+WW)}$$

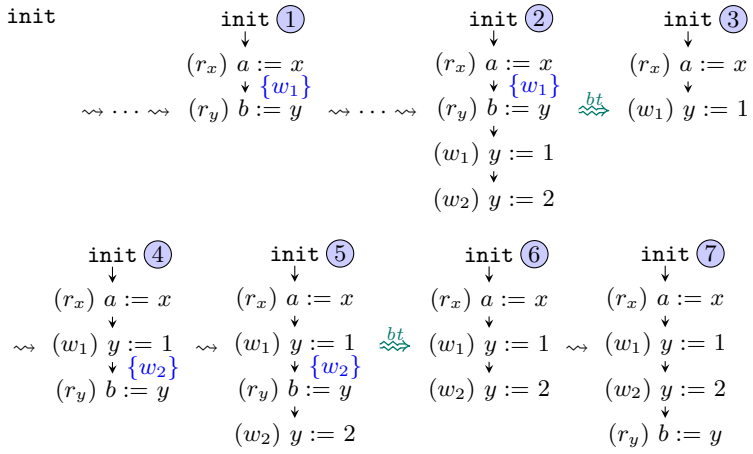

Fig. 1. Left-to-right DPOR exploration of RR+WW

After firing the transitions (r_x) and (r_y) (trace ①), DPOR adds transition (w_1) to the backtrack set of the state after the firing of transition (r_x), since transition (w_1) is in conflict with transition (r_y). When the initial exploration is finished (trace ②), DPOR backtracks to ① and considers the second exploration option, i.e., firing transition (w_1) and thus reaching ③.

Subsequently, DPOR fires (r_y) (trace ④) and notices that this is in conflict with (w_2); it then adds (w_2) as an alternative exploration option for the state before the firing of (r_y) in ④. Again, DPOR finishes with the exploration where the read instruction reads the value 1 (trace ⑤) and backtracks to ③. Now, (w_2) is fired (trace ⑥) and the algorithm continues with the remaining transition, leading to ⑦. DPOR now terminates since there is no other exploration option.

This way, DPOR manages to explores all three equivalence classes (representatives ②, ⑤, ⑦) of the 6 interleavings that correspond to this program.

2.2 Bounded Partial Order Reduction

Preemption bounding (PB) [26] prunes the state space by discarding executions that contain more preemptions than a given constant bound k. A preemption occurs at index i of a sequence of events τ whenever (1) events τ_i and τ_{i+1} originate from different threads and (2) the thread of τ_i remains enabled after τ_i; in particular, τ_i is not the last event of its thread.

Combining DPOR and PB is non-trivial. Specifically, simply pruning from DPOR's exploration space any trace with more than k preemptions is incorrect because their exploration might lead to exploring traces with up to k preemptions.

To see this, consider the run of RR+WW with $k = 0$. DPOR reaches the state where (r_x) is fired and (w_1) is considered as an alternative option in the backtrack set. Firing transition (w_1) will lead to trace ③, which exceeds the bound, since

there is a transition from the second thread present, while the first thread is still enabled. By discarding this state, the execution where $b = 2$ (which is equivalent to (7)) would never be considered, even though it respects the bound.

To address this issue, Coons et al. [10] conservatively add more backtrack points accounting for such bound-induced dependencies. Concretely, when the two transitions of the first thread are fired (trace (1)), Coons et al. [10] adds (w_1) in the backtrack set not only of the state before the firing of (r_y) in (2), as in the unmodified DPOR algorithm, but also of the initial state. Additionally, the initial transition from a state is always picked so that it is from the same thread as of the last fired transition, if possible. As a result, when the state with only (w_1) being fired is reached (due to the additional backtrack point), (w_2) will be fired immediately afterwards, and eventually the interleaving that corresponds to the right-to-left execution of the threads will be explored.

While this solution guarantees that no execution within the bound is lost, it weakens DPOR, i.e., it leads to the exploration of equivalent interleavings that would otherwise not be considered. In RR+WW, for $k > 0$, Coons et al. [10] explore interleavings that only differ in the order of (r_x) and (w_1).

2.3 TruSt: Optimal Dynamic Partial Order Reduction

The basic DPOR algorithm described in § 2.1 does not guarantee optimality, i.e., that only one execution from each equivalent class will be explored. There are several improvements of the basic algorithm, some of which achieve optimality (e.g., [2, 18]). Here, we follow the most recent such improvement, TruSt [16], which achieves optimality with polynomial memory consumption.

TruSt represents program executions as *execution graphs*, a concept that appeared in previous works for DPOR under weak memory models [15, 18]. An execution graph G consists of a set of nodes $G.\mathtt{E}$ (a.k.a. events) representing the individual thread instructions executed, such as read events `R` and write events `W`, and three kinds of directed edges encoding the ordering between events:

- the *program order* $G.\mathtt{po}$, which orders events of the same thread;
- the *coherence order* $G.\mathtt{co}$, which orders writes to the same location; and
- the *reads-from* mapping $G.\mathtt{rf}$, which shows where each read is reading from.

For an execution graph G, we define the following derived relations:

$$G.\mathtt{porf} \triangleq \left(G.\mathtt{po} \cup \left\{\langle G.\mathtt{rf}(r), r\rangle \middle|\, r \in G.\mathtt{R}\right\}\right)^{+} \qquad \text{(causality order)}$$

$$G.\mathtt{fr} \triangleq \left\{\langle r, w\rangle \middle|\, \langle G.\mathtt{rf}(r), w\rangle \in G.\mathtt{co}\right\} \qquad \text{(reads-before)}$$

The causality order, `porf`, relates two events if there is a path of program order or read-from dependencies between them, while `fr` orders a read event before every write that is coherence after the one read by the read.

An execution graph is SC-consistent (sequentially consistent) if there is a total ordering of its events respecting `po` such that each read event reads from the immediately preceding same-location write in the total order. Equivalently, a graph is SC-consistent if $\mathtt{porf} \cup \mathtt{co} \cup \mathtt{fr}$ is acyclic.

Execution graphs enable the efficient reversal of many conflicting events. If a write or a read event is in conflict with a previous write event, there is no need to backtrack to the state before the write events is added. Instead, the new event can be directly added in the execution and either read from a co-earlier write in case of a read event, or be placed co-before the conflicting write in case of a write event.

The only reversals where backtracking is necessary are those between a write event and a previously added read event: when a read event is added, it does not have the option to read from a write that has not yet been added. These reversals are referred to as *backward revisits*. To avoid exponential memory consumption, TruSt considers each exploration option eagerly when the new event is added, instead of maintaining backtrack sets for later exploration. In the case of backward revisits, TruSt removes the part of the execution that was added after the read event but is not in the *prefix* of the write event. The prefix of an event is defined as the set of events that precede it in the porf order. This allows the write event to be directly added in the execution graph. Because there is the possibility that many different execution graphs can lead to the same execution after a backward revisit, TruSt only considers the revisit if the events to be removed respect a *maximality condition* which is defined in such a way so that there will always be exactly one such set of deleted events, achieving an optimal exploration.

3 Bounded Optimal DPOR: Obstacles

We discuss the two main obstacles that complicate the application of preemption-bounded search to a DPOR algorithm.

3.1 Pessimistic Bound Definition

The first problem concerns the definition of preemptions for incomplete executions. Recall in the RR+WW example why the naive adaptation of DPOR with preemption bound $k = 0$ (incorrectly) does not generate the execution reading $b = 2$. The partial trace ③ is discarded because it contains at least one preemption according to the definition of Musuvathi et al. [24]. (Both threads are enabled and have executed one instruction each.)

We argue that this trace should be deemed to have no preemptions because of monotonicity. Trace ③ can be extended to a full trace (namely, ⑦) that (is equivalent to one that) does not have any preemptions.

We therefore modify the definition of preemptions as follows. A preemption occurs at index i of an event sequence τ whenever (1) events τ_i and τ_{i+1} originate from different threads and (2) the thread of τ_i remains enabled after τ_i, and has further events in the trace $\tau_{i+1}\tau_{i+2} \dots \tau_{|\tau|}$. According to our new definition, both interleavings that are equivalent with ③ have zero preemptions, because when switching to another thread, the first thread has no further events in the trace.

Our new definition satisfies monotonicity and coincides with the original on complete executions. We note, however, that partial executions with k preemptions

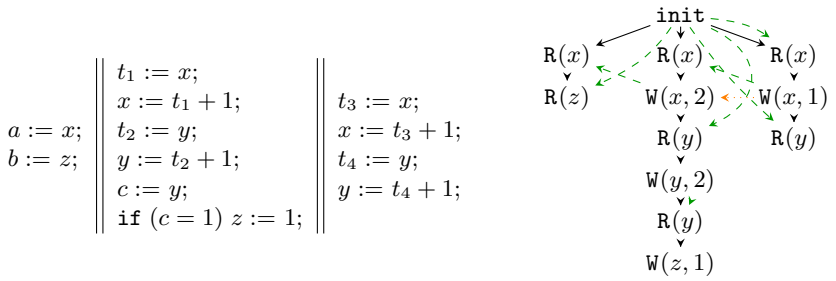

Fig. 2. A program and its intermediate execution that TruSt must explore in order to reach the right-to-left execution.

cannot always be extended to a complete execution with k preemptions. Consider, for example, trace ④ of RR+WW, which has no preemptions. Firing the only remaining transition leads to trace ⑤, which has one preemption. A DPOR algorithm that employs our definition of preemptions might thus reach states that are *bound-blocked*; the current explored execution respects the bound but there is no final execution reachable from this state that respects the bound. In our experience (see §6), bound-blocked executions do not seem to have a significant effect on the performance of our algorithm.

3.2 Need For Slack

Monotonicity alone is not enough to incorporate bounded search in an algorithm like TruSt, without still forfeiting completeness: some executions that respect the bound might still be lost. Intuitively, since DPOR algorithms operate by detecting conflicting instructions during an interleaving's exploration and reversing the conflict to obtain a new interleaving, it might be the case that for the conflict to be revealed, an execution that exceeds the bound needs to be explored.

We illustrate this point with the example in Fig. 2 where all the variables are initialized to zero. Consider a run of TruSt that always adds the next event from the left-most enabled thread. To reach the final execution that results from executing the threads from right to left, TruSt needs to pass through the execution depicted on the right of Fig. 2 before reaching this final execution. In the next step, the second write of the third thread will be added, which will reveal a conflict with the first read of y of the second thread. The algorithm will then perform a backward revisit, removing the events of the second thread after the first read of y, and change the read's incoming `rf` edge to the new write event. The desired final execution will be reached after the remaining events of the second thread are added again.

It is easy to see that, while the final execution has zero preemptions, the depicted intermediate execution has at least one preemption, and would thus be discarded. This example can in fact be generalized by adding more threads identical to the third one; to reach the final right-to-left execution that has zero preemptions, TruSt must visit an execution that has at least $N-2$ preemptions,

where N is the total number of threads. In §4, we show that this is in fact an upper limit; a final execution with k preemptions is always reachable through a sequence of executions that never exceed $k + N - 2$ preemptions. This result directly enables us to incorporate preemption-bounded search into TruSt by allowing some *slack* to the bound.

4 Recovering Completeness via Slack

Our bounded DPOR algorithm, Buster, can be seen in Algorithm 1, where we have highlighted the differences w.r.t. to TruSt [16].

We first discuss some additional notation used in the algorithm. First, each execution graph generated by the algorithm keeps track of the order $<_G$ in which events were added to it. Second, given a graph G and a set of events E, we write $G|_E$ for the restriction of G to E. Third, let $G.\mathsf{cprefix}(e)$ be the causal prefix of an event e in an execution graph G, i.e., the set of all events that causally precede it (including e itself). Formally, $G.\mathsf{cprefix}(e) \triangleq \left\{e' \middle| \langle e', e\rangle \in G.\mathtt{porf}^*\right\}$. Fourth, a subscript $\mathtt{loc}(a)$ restricts a set of events to those that access the same location as event a. Fifth, the function $\mathsf{SetRF}(G, a, w)$ adds an `rf` edge from w to a and $\mathsf{SetCO}(G, w_p, a)$ places a immediately after w_p in `co`. Finally, we define the *traces* of an execution graph as the linearizations of $(G.\mathtt{porf} \cup G.\mathtt{co} \cup G.\mathtt{fr})$ on $G.\mathtt{E}$. We lift the definition of preemptions to an execution graph G: $\mathtt{preemptions}(G)$ is the minimum number of preemptions in the traces of G.

Apart from only exploring SC-consistent executions, Buster eagerly discards executions with more preemptions than the user-provided value k plus the slack (Line 5). If both tests fail, Buster continues by picking an new event to extend the current execution (Line 6). For correctness, we fix $\mathsf{next}_P(G)$ to always return the event that corresponds to the left-most available thread. Depending on the type of the new event, the algorithm proceeds in a different way. We discuss the interesting cases of read and write events.

If the new event a is a read event, Buster simply considers every possible write event as an `rf` option for a (Line 13), and eagerly explores the corresponding execution. If a is a write event, first every `co` placement is considered and explored (Line 15). Afterwards, Buster considers possible backward-revisits; for every read r event that is not in the causal prefix of a, the execution where r reads from a is considered, after deleting the events added after r, that are not in the causal prefix of a (Line 19). To avoid redundant revisits, only when the set of deleted events satisfies a maximality condition (Line 18), is the backward-revisit performed (see [16] for more details).

4.1 Properties of TruSt

We now present some key properties of the TruSt algorithm, i.e., Algorithm 1 without Line 5, that are used to prove Buster's correctness (Theorem 1).

From TruSt's correctness argument, we know that every SC-consistent execution G_f has exactly one sequence of $\textsc{Visit}_P$ calls that leads to it. We call the sequence of the corresponding graphs a *production sequence* for G_f.

Algorithm 1 A Bounded DPOR algorithm based on TruSt [16]

```
1: procedure VERIFY(P, k)
2:     VISIT_{P,k}(G_∅)
3: procedure VISIT_{P,k}(G)
4:     if ¬consistent(G) then return
5:     if preemptions(G) > k + N − 2 then return
6:     switch a ← next_P(G) do
7:         case a = ⊥
8:             return "Visited full execution graph G"
9:         case a ∈ error
10:            exit("Visited erroneous execution graph G")
11:        case a ∈ R
12:            for w ∈ G.W_{loc(a)} do
13:                VISIT_{P,k}(SetRF(G, a, w))
14:        case a ∈ W
15:            VISITCOS_{P,k}(G, a)
16:            for r ∈ G.R_{loc(a)} \ G.cprefix(a) do
17:                Deleted ← {e ∈ G.E | r <_G e} \ G.cprefix(a)
18:                if ∀e ∈ Deleted ∪ {r}. ISMAXIMALLYADDED(G, e, a) then
19:                    VISITCOS_{P,k}(SetRF(G|_{G.E\Deleted}, r, a), a)
20:        case _
21:            VISIT_{P,k}(G)
22: procedure VISITCOS_{P,k}(G, a)
23:     for w_p ∈ G.W_{loc(a)} do VISIT_{P,k}(SetCO(G, w_p, a))
```

Given two SC-consistent graphs G and G', we say that G is a *prefix* of G', and write $G \sqsubseteq G'$, if $G'|_{G.\mathtt{E}} = G$. Intuitively, G is a prefix of G' if we can construct G' from G, by adding the missing events in some order for some `rf` and `co`.

Let a *maximal step* of an execution G be a execution that results from extending a thread of G by an event e in a *maximal* way, i.e., if $e \in \mathtt{R}$, then e is made to read from the `co`-latest event and if $e \in \mathtt{W}$, then e is placed at the end of `co`. We write $G \to G'$ when G' is a maximal step of G, and $G \to_e G'$ when $G \to G'$ and e is the added event. We say that a sequence of maximal steps is non-decreasing when the sequence of the thread identifiers of the added events is non-decreasing. Finally, we write $\mathtt{tid}(e)$ for the thread identifier of an event e.

A key property of TruSt (stated in Prop. 1) is that every execution G in the production sequence of an SC-consistent execution G_f is either a prefix of G_f, or it contains a read event r that does not read from the "correct" write, but there is a prefix $\hat{G}$ of G_f that can by extended to G by a non-decreasing sequence of maximal steps starting with r and not including events of at least one thread to the right of r.

Proposition 1. *Let S be the production sequence of an SC-consistent final execution G_f, and G be an execution in S. Then, either $G \sqsubseteq G_f$ or there ex-*

ists an execution G_b that is before G in S, a read event $r = \mathsf{next}_P(G_b)$, a thread $t > \mathtt{tid}(r)$ and an execution $\hat{G}$ such that $G_b \sqsubseteq \hat{G} \sqsubseteq G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(r)}$, $G_f|_{G_f.\mathsf{cprefix}(G_f.\mathtt{rf}(r))} \not\sqsubseteq G$, there is a non-decreasing sequence of maximal steps s.t. $\hat{G} \rightarrow_r \rightarrow^* G$, and $\forall e \in G.\mathtt{E} \setminus \hat{G}.\mathtt{E}.\, \mathtt{tid}(e) \neq t$.

Intuitively, TruSt tries to construct G_f by exploring an increasing sequence of its prefixes. This is not always possible, because when a read event r is added to G_b, the write event w that it should read from might not yet be present in G_b. In that case, r is made to read from another write and is later revisited by w leading to the execution $G'_b = G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(r)}$, which is a prefix of G_f. It is possible that additional backward revisit steps may happen between G_b and G'_b. Due to maximality, however, for every intermediate execution G in the production sequence between G_b and G'_b, there will be an execution $G_b \sqsubseteq \hat{G} \sqsubseteq G'_b$ that can be extended to G by a sequence of non-decreasing maximal steps. Execution $\hat{G}$ is exactly the part of G that is not deleted or revisited in a later step in S. Hence, if w is the first write that performed a backward revisit in S after G, then the events of thread $t = \mathtt{tid}(w)$ are already included in $\hat{G}$. Finally, it can be shown that t is to the right of r. The formal proof of this proposition can be found in the extended version of this paper [23].

4.2 Correctness of Slacked Bounding

To see why executions in the production sequence of a graph G_f can have at most $\mathtt{preemptions}(G_f) + N - 2$ preemptions, we start with a definition. A *witness* of a graph G is a trace of G that contains $\mathtt{preemptions}(G)$ preemptions.

Next, we observe that preemptions are monotone w.r.t. execution prefixes. That is, if an execution G requires a certain number of preemptions to be produced, a larger execution $G' \sqsupseteq G$ requires at least that many preemptions.

Lemma 1. *If G, G' are SC-consistent and $G \sqsubseteq G'$, then $\mathtt{preemptions}(G) \leq \mathtt{preemptions}(G')$.*

To prove this, take a witness of G' and restrict to the events of G, thereby obtaining a witness of G. The restriction can only remove preemptions.

Further, we note that the number of preemptions of an execution is unaffected if we extend its last executed thread with a maximal step; if a maximal step adds an event to a different thread, the number is increased by at most one.

Lemma 2. *Let G and G' be SC-consistent executions and $r \in G'.\mathtt{E}$ such that $G \rightarrow_r \rightarrow^* G'$. Then, $\mathtt{preemptions}(G') \leq \mathtt{preemptions}(G) + S$, where S is the number of threads that where extended to obtain G' from G.*

Proof. Consider a witness w of G and extend by appending the missing events in the same order they were added in the sequence of maximal steps. Notice that, by construction of the maximal step, the resulting sequencing is a trace of G'. Each time we add an event e in the trace, such that the last event of of the trace was not in the thread of e, we increase the preemption-bound by one: a thread

was previously considered as completed, but was now extended with a new event. However, this can only happen S times: the maximal steps keep adding events of the same thread and when another thread is picked, the first is not extended again (the maximal steps are non-decreasing). This gives us a trace of G' with at most $\texttt{preemptions}(G) + S$ preemptions, which concludes our proof. □

We can now prove that BUSTER is complete, i.e., it visits every full, SC-consistent execution that respects the bound.

Theorem 1. VERIFY(P, k) *visits every full, SC-consistent execution* G_f *of* P *with* $\texttt{preemptions}(G_f) \leq k$.

Proof. Consider a full, SC-consistent execution G_f of P with at most k preemptions. From the completeness of TruSt, we know that a run of Algorithm 1 without the test on Line 5 will visit G_f. It thus suffices to show that for every execution G in the production sequence of G_f has at most $k + N - 2$ preemptions, where N is the number of threads of P. If $G \sqsubseteq G_f$, then from Lemma 1 $\texttt{preemptions}(G) \leq \texttt{preemptions}(G_f) \leq k$.

Otherwise, from Prop. 1, there exists an execution G_b that is before G in the production sequence of G_f and an execution $\hat{G}$, such that $G_b \sqsubseteq \hat{G} \sqsubseteq G_f|_{G_b.\texttt{E} \cup G_f.\mathsf{cprefix}(r)}$, $\mathsf{next}_P(G_b) = r \in \texttt{R}$, $G_f|_{G_f.\mathsf{cprefix}(G_f.\texttt{rf}(r))} \not\sqsubseteq G$, $\hat{G} \rightarrow_r \rightarrow^* G$, and no events in $G.\texttt{E} \setminus \hat{G}.\texttt{E}$ are in thread t, for some thread t to the right of r.

From the last two properties and Lemma 2 we have $\texttt{preemptions}(G) \leq k+N-1$ since it is $\texttt{preemptions}(\hat{G}) \leq \texttt{preemptions}(G_f)$ ($\hat{G} \sqsubseteq G_f$ and Lemma 1) and at most $N-1$ threads are extended from $\hat{G}$ to G.

To complete the proof, we will prove that $\texttt{preemptions}(G) = k + N - 1$ leads to contradiction. The equality implies that $\hat{G}$ had k preemptions and that $N-1$ threads were extended in the maximal steps from $\hat{G}$ to G, and all of them increased the preemptions by one. The sequence of maximal steps from $\hat{G}$ to G is non-decreasing and starts with the thread of r. Since there are at most N threads, $N-1$ are extended, and at least one thread to the right of t is not extended, r is in the leftmost thread.

Let t_r be the leftmost thread, $G'_b \triangleq G_f|_{G_b.\texttt{E} \cup G_f.\mathsf{cprefix}(r)}$, and $w \triangleq G_f.\texttt{rf}(r)$. From the proof of TruSt, we can infer that all events of G_b are in the $\texttt{porf}$-prefix of the last event of t_r. It is $G_f|_{G_f.\mathsf{cprefix}(w)} \not\sqsubseteq G_b$: the opposite, together with $G_b \sqsubseteq \hat{G} \sqsubseteq G$, contradicts $G_f|_{G_f.\mathsf{cprefix}(w)} \not\sqsubseteq G$. Since G_b is in the production sequence of G_f, $G_b \sqsubseteq G_f$, $\mathsf{next}_P(G_b) = r$, and $G_f|_{G_f.\mathsf{cprefix}(w)} \not\sqsubseteq G_b$, TruSt will eventually add the write $w \triangleq G_f.\texttt{rf}(r)$ and revisit the read r, reaching the execution $G'_b \sqsubseteq G_f$ that contains all events added before r, i.e., the events of G_b, the events in the $\texttt{porf}$-prefix of r, and r. Hence, all events in $G'_b.\texttt{E} \setminus \{r\}$ are in the $\texttt{porf}$-prefix of r, which implies that any witness of G'_b ends with r.

Since $G'_b \sqsubseteq G_f$, any witness t of G'_b has at most k preemptions. Let G' be the execution G'_b without r, and G'' the unique execution s.t. $\hat{G} \rightarrow_r G''$. Removing the last event r from t gives us a trace t' of G' with at most k preemptions. If t' ends with an event of t_r, then we can restrict t' to the events of $\hat{G}$ and add r at the end, obtaining a trace of G'' with at most k preemptions. Otherwise, t' does

not end with an event of t_r, and thus trace t has one more preemption than t, i.e., t' has at most $k-1$ preemptions. Then, we can again restrict t' to the events of $\hat{G}$ and add r a the end, obtaining again a trace of G'' with at most k preemptions. This contradicts our assumption that $\texttt{preemptions}(\hat{G}) = k$ and all $N-1$ threads that are extended from $\hat{G}$ increase the number of preemptions, since the first thread t_r can be extended without incurring any more preemptions. □

Buster inherits TruSt's optimality, as it only explores a subset of the executions that TruSt does. Here, optimality refers to avoiding redundant work; due to the slack, Verify(P, k) can also visit executions more than k preemptions.

Theorem 2. Verify(P, k) *explores each graph G of a program P at most once.*

5 Implementation

We have implemented Buster on top of the GenMC tool [19], which implements the TruSt algorithm [16]. Since GenMC supports weak memory models and the standard notion of preemption bounding only makes sense for sequential consistency, we enforce SC in our benchmarks by using only SC memory accesses and selecting GenMC's RC11 model [21].

The bulk of our modifications to GenMC concern the checking of whether the preemption-bound of an execution G exceeds a value k. Generally, deciding whether the preemption-bound of a Mazurkiewicz trace exceeds a value is an NP-complete problem [24]. We use an adaptation of the bound computation in Musuvathi et al. [24] to execution graphs, but instead of recursively computing $\texttt{preemptions}(G)$ (and cache computations across calls to amortize the cost), we recursively compute the predicate $\Phi(G, k) \triangleq \texttt{preemptions}(G) \leq k$. The benefit of this method is that we can avoid calculating $\texttt{preemptions}(G)$ exactly when its value exceeds the desired bound. Furthermore, there is no additional state that needs to be stored; Buster remains stateless.

As an optimization, we use as slack (Line 5) the minimum between $N-2$ and the number of threads that have no *deletable* events; an event is not deletable if it is in the porf-prefix of a write that backward revisited. Intuitively, the events that are added in G to reach $\hat{G}$ (Prop. 1) are the events that will later be deleted to eventually reach a graph that is a prefix of the final graph G_f.

6 Evaluation

To evaluate Buster, we answer the following questions:

- § 6.1 How many preemptions suffice to expose common concurrency bugs? Is Buster effective at finding such concurrency bugs?
- § 6.2 How good is preemption bounding at pruning the search space? Up to what bound does Buster run faster than vanilla DPOR?
- § 6.3 What is the overhead induced by the bound calculation?

Table 1. Buggy benchmarks. An ✗ indicates that an error was found.

	$k=0$		$k=1$		$k=2$		GenMC	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time
account-bad	3 ✗	0.01	3 ✗	0.01	3 ✗	0.01	3 ✗	0.01
bluetooth-driver-bad	1	0.01	3 ✗	0.02	7 ✗	0.02	8 ✗	0.01
circular-buffer-bad	2	0.07	13 ✗	0.49	1 ✗	0.03	1 ✗	0.03
din-phil-sat	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
fsbench-bad	0 ✗	0.93	0 ✗	0.93	0 ✗	0.94	0 ✗	1.01
lazy01-bad	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
queue-bad	20	1.91	56 ✗	27.47	2 ✗	0.18	2 ✗	0.19
reorder-20-bad	🕒	🕒	🕒	🕒	🕒	🕒	10 ✗	0.05
stack-bad	11	0.44	10 ✗	0.35	10 ✗	0.35	10 ✗	0.37
token-ring-bad	12 ✗	0.02	12 ✗	0.02	12 ✗	0.02	12 ✗	0.02
twostage-100-bad	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
wronglock-bad	5914	164.46	2 ✗	0.02	2 ✗	0.02	2 ✗	0.02
lazy01-unsafe	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
sigma-unsafe	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
singleton-unsafe	5 ✗	0.01	5 ✗	0.01	5 ✗	0.01	5 ✗	0.01
stateful01-1-unsafe	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01	0 ✗	0.01
triangular-2-unsafe	6	0.04	66	0.40	368	2.06	9069 ✗	29.44
stack-2-unsafe	6	0.06	5 ✗	0.05	5 ✗	0.05	5 ✗	0.05
read-write-lock-2-unsafe	68	0.51	53 ✗	0.25	132 ✗	0.59	276 ✗	0.96
reorder-2	417	0.14	6 ✗	0.01	2 ✗	0.01	2 ✗	0.01

§ 6.4 What is the overhead induced by bound-blocked executions?

To that end, we evaluate Buster against GenMC on a diverse set of benchmarks. Unfortunately, we cannot include the approach of Coons et al. [10] in our comparison because their implementation is not available.

We can draw two major conclusions from our evaluation. First, most bugs do manifest with a small number of preemptions (≤ 2), an observation that has been made in the literature before [26, 28]. Second, even though the bound calculation can be fairly expensive expensive, for small bounds Buster outperforms GenMC and can find bugs faster than GenMC.

Experimental Setup We conducted all experiments on a Dell PowerEdge M620 blade system with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz) and 256GB of RAM. We used LLVM 11.0.1 for GenMC and Buster. All reported times are in seconds. We set a timeout limit of 30 minutes.

6.1 Bound and Bug Manifestation

To validate that most bugs require a small number of preemptions, we run Buster and GenMC on three sets of benchmarks:

- the unsafe concurrent benchmarks of the SCT suite [28],
- the unsafe benchmarks of the pthread category of SV-COMP [27] included in GenMC's test suite, and
- a set of concurrent data structures (CDs) from GenMC's test suite with randomly induced bugs.

In all cases, we configure Buster to disregard any errors that occur in executions that exceed the bound and are explored due to the slack. We note that this

Table 2. Buggy CD benchmarks. An ✗ indicates that the error was found.

	$k=0$		$k=1$		$k=2$		GenMC	
Benchmark	Execs	Time	Execs	Time	Execs	Time	Exec	Time
dglm-queue-bug(6)	48 ✗	2.55	305 ✗	102.25	810 ✗	272.71	🕒	🕒
dglm-queue-bug(7)	54 ✗	3.94	404 ✗	209.22	1259 ✗	628.52	🕒	🕒
dglm-queue-bug(8)	60 ✗	5.88	517 ✗	393.02	1854 ✗	1320.58	🕒	🕒
ms-queue-bug(6)	84 ✗	7.71	1366 ✗	155.08	9906 ✗	1057.28	🕒	🕒
ms-queue-bug(7)	103 ✗	12.87	1936 ✗	294.76	🕒	🕒	🕒	🕒
ms-queue-bug(8)	124 ✗	20.72	2636 ✗	530.04	🕒	🕒	🕒	🕒
bstack(7)	2	0.24	19 ✗	1.26	83 ✗	3.55	🕒	🕒
bstack(8)	2	0.34	22 ✗	2.06	111 ✗	6.41	🕒	🕒
bstack(9)	2	0.48	25 ✗	3.23	143 ✗	10.95	🕒	🕒
msq-bug2(5)	2	0.09	18 ✗	0.48	154 ✗	2.69	37420 ✗	280.64
msq-bug2(6)	2	0.12	22 ✗	0.87	232 ✗	6.29	🕒	🕒
stack-oe-bug(4)	77	0.64	1086	17.77	375 ✗	9.66	3523 ✗	97.65
stack-oe-bug(5)	92	1.04	1700	38.25	663 ✗	23.61	17032 ✗	763.96
stack-oe-bug(6)	107	1.58	2478	74.83	1076 ✗	50.38	🕒	🕒
stack-oe-bug(7)	122	2.32	3435	134.89	1638 ✗	97.52	🕒	🕒

configuration may delay bug finding, since Buster may by chance quickly come across a buggy execution with more than k preemptions (due to slack) before finding any buggy execution with up to k preemptions. Nevertheless, we follow it to ensure that the bugs found arise in executions with up to the desired number of preemptions, so as to be able to validate the claim that bugs manifest in executions with a small number of preemptions.

Table 1 reports our outcomes on the first two classes of benchmarks. As can be seen, Buster was able to find most bugs using a bound of 1. In fact, for most benchmarks, Buster found the bug before exploring a complete execution, hence the "0 ✗" entries in the table. The only benchmarks, where Buster needs a bound greater that 1 are the synthetic benchmarks `triangular`, which needs a bound of 8, as it was specifically designed to make the bug discovery difficult and push model checkers to their limits; `reorder-20` and `twostage-100`, which have a large number of threads (20 and 100, respectively). Buster times out on the latter two benchmarks because the large number of threads put a lot of stress in the bound checking procedure. We note that for `twostage-100`, GenMC also fails to terminate within the time limit.

Table 2 reports our results for our CD benchmarks. For these benchmarks, we have taken CD implementations from the GenMC test suite, and induced bugs into them by randomly dropping a synchronization instruction or replacing a CAS instruction with a normal write or an unconditional exchange instruction, thereby introducing a possible atomicity violation. We then construct medium-sized clients (with 2-3 threads and up to 12 operations per thread) of these data structures that check for their intended semantics (for example, that a queue has FIFO semantics). In all cases, the induced bugs lead to violations of the assertions in the client programs, and occasionally even to memory errors. Buster can find these bugs easily; a bound of $k = 2$ suffices to expose them. By contrast, GenMC times out for most of these benchmarks, as their state space is enormous.

Table 3. Buster and GenMC comparison on safe data structure benchmarks.

	$k=0$		$k=1$		$k=2$		$k=3$		GenMC		Max
Benchmark	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	k
dglm-queue(6)	2	0.61	12	3.05	62	11.30	162	27.14	924	104.47	7
dglm-queue(7)	2	0.97	14	5.78	86	25.65	266	71.73	3432	570.68	8
ms-queue(6)	2	0.30	18	2.23	128	8.46	513	29.46	18564	321.58	8
ms-queue(7)	2	0.46	21	4.16	177	18.53	840	78.13	◷	◷	
bstack2(8)	2	0.12	16	0.58	114	2.97	408	9.17	12870	159.27	9
bstack2(9)	2	0.15	18	0.88	146	5.08	594	17.75	48620	720.06	8
bstack(5)	2	0.12	20	0.53	92	2.98	310	7.87	4214	88.01	8
bstack(6)	2	0.18	24	0.97	134	6.84	549	21.35	26040	787.64	8
ms-queue(7)	2	0.19	14	1.19	86	5.77	266	16.41	3432	135.85	7
ms-queue(8)	2	0.26	16	1.85	114	10.29	408	33.78	12870	641.64	8
stack-oe(4)	77	0.64	1098	17.62	6208	139.81	23472	641.13	◷	◷	
stack-oe(5)	92	1.06	1713	39.55	11510	377.50	◷	◷	◷	◷	
ms-oe(6)	12	0.27	84	2.93	615	18.82	2039	57.58	10880	218.86	5
ms-oe(7)	14	0.34	100	3.97	800	27.42	2855	91.54	20823	458.09	5
dglm-oe(7)	5	0.20	29	2.14	129	9.27	238	19.53	248	20.88	3
dglm-oe(8)	5	0.23	31	2.62	146	11.77	294	26.33	306	28.50	3
dglm-fifo(7)	26	4.50	128	21.84	128	25.93	128	25.12	128	22.92	1
dglm-fifo(8)	29	6.81	162	35.43	162	42.66	162	41.59	162	37.91	1
ttas-lock2(7)	2	0.12	14	0.48	86	1.89	266	4.57	3432	28.50	7
ttas-lock2(8)	2	0.17	16	0.81	114	3.66	408	10.14	12870	121.94	8
ttas-lock3(4)	21	0.89	195	7.12	1041	29.94	3525	84.55	34650	387.36	5
ttas-lock3(5)	26	2.32	320	23.97	2274	130.62	10494	492.89	◷	◷	

6.2 Comparison with Plain DPOR on Safe Benchmarks

We have already seen that modulo specially crafted synthetic benchmarks, a small preemption bound is sufficient for finding bugs in practice. Moreover, Buster is pretty good at finding such bugs in concurrent data structures. We now evaluate the application of Buster on a collection of safe benchmarks. For this purpose, we use different variations of the benchmarks of Table 2 (after repairing them so that no assertion is violated), as well as a few locking benchmarks.

Table 3 compares the performance of Buster for small values of k and GenMC. As it can be seen, GenMC struggles with these benchmarks, whereas Buster with $k = 2$ (and often also with $k = 3$) terminates fairly quickly. This is because only a small fraction of the total executions of sizeable benchmarks have few preemptions. Therefore restricting the search to only those executions makes Buster run much faster than GenMC, and guarantees that the program under consideration does not have any common bugs.

In the last column of Table 3 we include the maximum value of k such that Buster terminates faster than GenMC, for the benchmarks that terminate under GenMC. In most cases Buster is faster than GenMC even for $k > 3$. For the `dglm-fifo` benchmarks Buster is only faster for $k \in \{0, 1\}$, because for these benchmarks a small k suffices to fully explore the state space.

6.3 Bound Calculation Overhead

We now measure the cost of checking that each encountered execution is below the specified bound. As we discussed in §5, checking whether an execution graph's preemption-bound exceeds a value is a NP-complete problem, and thus we expect this calculation to threaten the performance of our tool.

Table 4. Overhead w.r.t. to GenMC (left) and blocking in benchmarks (right).

Benchmark	b	$k=b$	$k=b+1$	$k=b+2$	GenMC
treiber(6,0)	0	10%	6%	4%	30.81
treiber(7,0)	0	23%	12%	5%	529.42
treiber(3,2)	1	6%	5%	5%	2.75
treiber(3,3)	1	7%	6%	5%	31.15
treiber(3,4)	1	13%	8%	6%	332.76
treiber(4,2)	1	9%	7%	5%	47.50
treiber(4,3)	2	10%	7%	5%	777.44
ttas-lock(6)	0	20%	13%	11%	14.52
ttas-lock(7)	0	38%	25%	16%	231.91

# Blocked	# Benchmarks
0	72
1	143
2	45
3	3
4	14
5	4
6	1
8	69
>8	6

To carefully account for this cost, we compare Buster against the baseline GenMC implementation on benchmarks where preemption bounding does not reduce the number of executions that are explored. In Table 4, we report results on simple CD clients that have only one operation per thread of the Treiber stack [29] and the TTAS lock [13]. The clients are designed so that Buster can explore the full set of program executions with a small bound k. We suffix the name of the benchmarks with the number of writer and reader threads for the Treiber stack and the total number of threads for TTAS.

Column b contains the minimal number of the bound k for which Buster explores the same number of executions as GenMC does. Note that since these benchmarks contain several threads, exploration up to a certain bound (e.g., $k = 0$) does not mean that only executions with k preemptions are visited; due to slack, executions with more preemptions may be visited, and so it is possible for the exploration to cover the entire state space for a smaller bound than intrinsically necessary. In the subsequent columns we report the time overhead (percentage) for bounds $k = b, k = b + 1$, and $k = b + 2$ w.r.t. to GenMC's execution time, which is visible on the last column. The maximum overhead is observed for $k = b$ (the minimal value sufficient to cover the entire state space). This is expected because $k = b$ places the most burden on the calculation of whether the number of preemptions in a given execution are below k. For larger k values, the overhead drops because it is easier to show that the number of preemptions are below the bound; one does not have to calculate the number of preemptions of an execution precisely. Overall, for the Treiber stack benchmark, the overhead introduced by calculating the bounds is fairly low and does not exceed the 23% of the execution time of GenMC. For the plain runs of ttas-lock, the maximal overhead is a bit larger, up to 38%. We note, however, that such overhead only occurs in clients with a large number of threads (7); smaller clients are not affected as much.

6.4 Overhead due to Bound-Blocked Executions

Finally, we measure the overhead caused by bound-blocked executions, by evaluating how often they arise in practice. Specifically, we ran Buster on GenMC's test suite for various preemption-bound values, as well as on the safe CD clients used in § 6.2, and counted the number of such bound-blocked executions.

For GenMC's test suite, the results are summarized in Table 4 (right). We have restricted out attention to the runs with at least 10 executions, so that our results are not skewed by benchmarks that have very few executions. We have also excluded 8 benchmarks from the test suite that use barriers because they are currently not supported by our tool. As it can be seen, bound-blocked executions are rare: most runs lead to one bound-blocked execution, and only 6 lead to more than 8 bound-blocked executions. Bound-blocked executions are on average no more than 6% of the total number of executions explored.

For the CDs clients, bound-blocked executions are even more rare; out of the 22 clients, Buster encounters bound-blocked executions in only 4 of them, for some k. We exclude again from the discussion runs with very few executions. From the remaining runs, only two encounter a considerable number of bound-blocked executions that become negligible as the bound is increased: around 10% for $k = 1$ and less than 1% for $k = 2$

7 Related Work

There is a large body of work that has improved the original DPOR algorithm of Flanagan et al. [11]. Abdulla et al. [2] introduced the first optimal DPOR algorithm, which, however, suffers from possibly exponential memory consumption. Kokologiannakis et al. [16] developed TruSt, which is the first optimal DPOR algorithm that consumes polynomial memory.

Agarwal et al. [6], Chalupa et al. [8], Chatterjee et al. [9], and Huang [14] have extended DPOR for partitions coarser than the one we have focused in this paper, i.e., Mazurkiewicz traces. Abdulla et al. [1, 4, 5] consider DPOR under various weak memory models, while the works of Kokologiannakis et al. [16, 18, 20] provide a DPOR algorithm that is parametric in the choice of the memory model, provided it respects some basic properties.

Qadeer et al. [26] showed the decidability of context-bound verification of concurrent boolean programs. Musuvathi et al. [25] propose *iterative* context bounding, a search algorithm that prioritizes executions with fewer preemptions. Musuvathi et al. [24] combine partial-order reduction with a preemption-bound search, and prove that judging whether the preemption-bound of a Mazurkiewicz trace exceeds a certain value is an NP-complete problem.

To our knowledge, the only attempt to combine DPOR and preemption bounding is by Coons et al. [10], who identify the difficulty of maintaining completeness of the exploration, and resolve it by weakening DPOR.

Abdulla et al. [3] and Atig et al. [7] have extended the notion of preemption bounding to weak memory models. We leave a possible extension of our approach to weak memory models for future work.

Acknowledgments We thank the anonymous reviewers for their valuable feedback. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

8 Data-Availability Statement

All supplementary material is available at [23]. The artifact is also available at [22].

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. "Stateless model checking for TSO and PSO". In: *TACAS 2015*. Vol. 9035. LNCS. Berlin, Heidelberg: Springer, 2015, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28. URL: http://dx.doi.org/10.1007/978-3-662-46681-0_28.

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

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. "Context-Bounded Analysis for POWER". In: *TACAS 2017*. Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 56–74. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_4.

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

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. "Optimal stateless model checking under the release-acquire semantics". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 135:1–135:29. ISSN: 2475-1421. DOI: 10.1145/3276505. URL: http://doi.acm.org/10.1145/3276505.

[6] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. "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, July 2021, pp. 341–366. ISBN: 978-3-030-81685-8. DOI: 10.1007/978-3-030-81685-8_16.

[7] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. "Context-Bounded Analysis of TSO Systems". In: *FPS 2014*. Ed. by Saddek Bensalem, Yassine Lakhneck, and Axel Legay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 21–38. ISBN: 978-3-642-54848-2. DOI: 10.1007/978-3-642-54848-2_2.

[8] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. "Data-centric dynamic partial order reduction". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 31:1–31:30. ISSN: 2475-1421. DOI: 10.1145/3158119. URL: http://doi.acm.org/10.1145/3158119.

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

[10] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. "Bounded Partial-Order Reduction". In: *OOPSLA 2013*. Indianapolis, Indiana, USA: ACM, 2013, pp. 833–848. ISBN: 9781450323741. DOI: 10.1145/2509136.2509556. URL: https://doi.org/10.1145/2509136.2509556.

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

[12] Patrice Godefroid. "Model checking for programming languages using VeriSoft". In: *POPL 1997*. Paris, France: ACM, 1997, pp. 174–186. DOI: 10.1145/263699.263717. URL: http://doi.acm.org/10.1145/263699.263717.

[13] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. 2008.

[14] Jeff Huang. "Stateless model checking concurrent programs with maximal causality reduction". In: *PLDI 2015*. New York, NY, USA: ACM, 2015, pp. 165–174. DOI: 10.1145/2737924.2737975. URL: http://doi.acm.org/10.1145/2737924.2737975.

[15] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. "Effective stateless model checking for C/C++ concurrency". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 17:1–17:32. ISSN: 2475-1421. DOI: 10.1145/3158105. URL: http://doi.acm.org/10.1145/3158105.

[16] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. "Truly stateless, optimal dynamic partial order reduction". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498711. URL: https://doi.org/10.1145/3498711.

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

[18] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. "Model checking for weakly consistent libraries". In: *PLDI 2019*. New York, NY, USA: ACM, 2019. DOI: 10.1145/3314221.3314609.

[19] Michalis Kokologiannakis and Viktor Vafeiadis. "GenMC: A model checker for weak memory models". In: *CAV 2021*. Ed. by Alexandra Silva and

K. Rustan M. Leino. Vol. 12759. LNCS. Springer, 2021, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20.

[20] Michalis Kokologiannakis and Viktor Vafeiadis. "HMC: Model checking for hardware memory models". In: *ASPLOS 2020*. ASPLOS '20. Lausanne, Switzerland: ACM, 2020, pp. 1157–1171. ISBN: 9781450371025. DOI: 10.1145/3373376.3378480. URL: https://doi.org/10.1145/3373376.3378480.

[21] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing sequential consistency in C/C++11". In: *PLDI 2017*. Barcelona, Spain: ACM, 2017, pp. 618–632. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062352. URL: http://doi.acm.org/10.1145/3062341.3062352.

[22] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. "Reconciling Preemption Bounding with DPOR (artifact)". In: (Apr. 2023). DOI: 10.5281/zenodo.7505917.

[23] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. "Reconciling Preemption Bounding with DPOR (supplementary material)". In: (Apr. 2023). URL: https://plv.mpi-sws.org/genmc.

[24] Madalan Musuvathi and Shaz Qadeer. *Partial-Order Reduction for Context-Bounded State Exploration*. Tech. rep. MSR-TR-2007-12. Microsoft Research, 2007. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2007-12.pdf.

[25] Madanlal Musuvathi and Shaz Qadeer. "Iterative Context Bounding for Systematic Testing of Multithreaded Programs". In: *PLDI 2007*. San Diego, California, USA: ACM, 2007, pp. 446–455. ISBN: 9781595936332. DOI: 10.1145/1250734.1250785. URL: https://doi.org/10.1145/1250734.1250785.

[26] Shaz Qadeer and Jakob Rehof. "Context-Bounded Model Checking of Concurrent Software". In: *TACAS 2005*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. LNCS. Springer, 2005, pp. 93–107. DOI: 10.1007/978-3-540-31980-1_7. URL: https://doi.org/10.1007/978-3-540-31980-1%5C_7.

[27] SV-COMP. *Competition on Software Verification (SV-COMP)*. 2019. URL: https://sv-comp.sosy-lab.org/2019/ (visited on 03/27/2019).

[28] Paul Thomson, Alastair F. Donaldson, and Adam Betts. "Concurrency testing using schedule bounding: an empirical study". In: *PPoPP 2014*. ACM, 2014, pp. 15–28. DOI: 10.1145/2555243.2555260. URL: https://doi.org/10.1145/2555243.2555260.

[29] R. Kent Treiber. *Systems Programming: Coping with Parallelism*. Tech. rep. Technical Report RJ5118, IBM, 1986. URL: https://dominoweb.draco.res.ibm.com/58319a2ed2b1078985257003004617ef.html.

A Proof of Prop. 1

We adopt the definitions and notations used in [17]. More specifically, we write $G \overset{e}{\underset{t}{\Rightarrow}} G'$ when a call to $\textsc{Visit}(P, G)$ directly leads to a call to $\textsc{Visit}(P, G')$, $\mathsf{next}_P(G) = e$, G' is SC-consistent, and t is the step taken to reach G': $t = rv\ r$ when the call was due to a revisit of a read r, otherwise $t = nr$. Additionally, we assume a total order $<_{\mathsf{next}}$ on the events of the program that respects $\mathtt{po}$.

Lemma 3. *If the algorithm reaches an execution G and a write event w is added in a revisit step, then w cannot be deleted in any later step.*

Proof. Follows directly from Lemma A.21 in [17]. □

Lemma 4. *Let S be the unique production sequence that reaches an SC-consistent final execution G_f and G be an execution in S such that $G \sqsubseteq G_f$. Then there is no step in S after G that revisits a read event of G.*

Proof. Suppose that there is, and let G' be the execution that results from that revisit step. If there are multiple such steps, we pick the one that revisits the $<$-earliest event, and among those the first step that appears in S. Thus we have that (a) $G \Rightarrow^* \overset{w}{\underset{rv\ r}{\Rightarrow}} G' \Rightarrow^* G_t$, where $r \in G.\mathtt{E}$, (b) for all revisit steps between G and G' that revisit an event $r' \in G.\mathtt{E}$, it is $r' > r$, and (c) for all revisit steps after G' that revisit an event $r' \in G.\mathtt{E}$, it is $r' >= r$.

Let $w \triangleq G.\mathtt{rf}(r)$. We will show that there is not step between G and G' that deletes w. Assume the opposite, i.e., there is a step t that revisits a read r' and deletes w. We assume that t is the first such step, i.e., the first step after G (and before G') that deletes w. From the hypothesis, it is $r' >= r$. Since w is deleted, it must be $w > r'$, and thus $w > r$. In the execution before t it must be $\mathtt{rf}(r) = w$, since t is the first step after G that deletes w, no step between G and G' deletes r, and $G.\mathtt{rf}(r) = w$. Thus t deletes w but not the read r that reads from w, which is a contradiction: even if t revisits r (i.e., $r' = r$), it cannot be that r was reading from a write that is deleted.

Therefore, no step between G and G' deletes w, which implies that $w \in G'.\mathtt{E}$, but $G'.\mathtt{rf}(r) = w'$. If $w < r$, it cannot be that a subsequent execution has $\mathtt{rf}(r) = w$ because no subsequent execution revisits a read $r' < r$ that could delete w. Thus, it is $w > r$ and $\langle w, w' \rangle \in G.\mathtt{porf}$. However, the write event w' cannot be deleted in a later step (Lemma 3), and thus neither can w, which implies that it cannot be that r reads from w in a later execution: neither r nor w will be deleted in a step after G' and $G'.\mathtt{rf}(r) = w' \neq w$. □

Proposition 1. Let S be the production sequence of an SC-consistent final execution G_f, and G be an execution in S. Then, either $G \sqsubseteq G_f$ or there exists an execution G_b that is before G in S, a read event $r = \mathsf{next}_P(G_b)$, a thread $t > \mathtt{tid}(r)$ and an execution $\hat{G}$ such that $G_b \sqsubseteq \hat{G} \sqsubseteq G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(r)}$, $G_f|_{G_f.\mathsf{cprefix}(G_f.\mathtt{rf}(r))} \not\sqsupseteq G$, there is a non-decreasing sequence of maximal steps s.t. $\hat{G} \rightarrow_r \rightarrow^* G$, and $\forall e \in G.\mathtt{E} \setminus \hat{G}.\mathtt{E}.\, \mathtt{tid}(e) \neq t$.

Proof. Assume $G \not\sqsubseteq G_f$, and let G_b be the last execution before G in S that is a prefix of G_f. Then it is $\mathsf{next}_P(G_b) = r \in \mathtt{R}$. To see this, assume that $\mathsf{next}_P(G_b)$ is not a read event and let G' be the execution immediately after G_b in S. From Lemma 4, no event of G_b can be revisited or deleted. This implies that $G' \sqsubseteq G_f$, which contradicts that G_b is the last execution in S before G that is a prefix of G_f. Assume now that $G_f|_{G_f.\mathsf{cprefix}(G_f.\mathtt{rf}(r))} \sqsubseteq G$. Then r reads from a write $w' \neq G_f.\mathtt{rf}(r)$ in G, and S will never reach G_f because both r and $G_f.\mathtt{rf}(r)$ cannot be deleted in a later step.

Because S ends in G_f, the is an execution after G in S that is a prefix of G_f. Let G'_b be the first such execution. From Lemma 4 is must be that $G'_b = G_f|_{G_b.\mathtt{E} \cup G_f.\mathsf{cprefix}(()r)}$: r is in G'_b since no event of G_b is ever revisited, it must read from $w = G_f.\mathtt{rf}(r)$, and any event other than $G_b \cup \{r\}$ is not in G'_b since they were deleted from the revisit of r from w.

Let S_b be the subsequence of S from G_b to G'_b. For each execution $\bar{G}$ in S_b, we define as $P(\bar{G})$ the set of events in $\bar{G}.\mathtt{E} \setminus G_b.\mathtt{E}$ that are not revisited or deleted in a later step until G'_b in S. Similarly, we define ad $G(\bar{G})$ the set of events in $\bar{G}$ that are revisited or deleted in a later step until G'_b in S. It is easy to see that any event in $P(\bar{G})$ is in the $\mathtt{porf}$-prefix of w in G'_b; otherwise, it would be deleted by the last step of S_b, which revisits r from w.

We will show that any event $d \in D(G)$ is maximal in G w.r.t. the set $P(G) \cup B$, where B is the set of events added before d in G. Assume the contrary, i.e., $d \in D(G)$ is first revisited or deleted in a later $t = \overset{w'}{\underset{rv\ r'}{\Rightarrow}}$ step of S_b that results in an execution G'. Observe that no event of B is revisited or deleted before t in S_b: a revisit from a write w'' to an event $d' \in D(G)$ with $d' \neq d$ would either also delete d or imply that d is in the prefix of w'', which would disallow the revisit step t (Lemma 3). Then the revisit step t cannot have happened: d is not maximal in G w.r.t. to the set $P(G) \cup B$ and the algorithm requires that d is maximal in a suffix of G w.r.t. to a set that contains $P(G) \cup B$.

We will show that for any execution $\bar{G}$ in S_b and any pair $\langle a, b\rangle$ of events of $D(\bar{G})$, if $a <_{\mathsf{next}} b$, then $a < b$. Assume the contrary and let G' be the first execution with $\{a, b\} \in D(\bar{G})$, $a <_{\mathsf{next}} b$, and $a > b$. From [17], we have that b is in the $\mathtt{porf}?$-prefix of a write that revisited, which contradicts that b is revisited or deleted in a later step ($b \in D(\bar{G})$). Read r is the first event of $D(\bar{G})$ in addition order because no event added before r is revisited (Lemma 4). Additionally, there is no event $d' \in D(\bar{G})$ that is $d' <_{\mathsf{next}} r$, since this would implies that $d' < r$.

By defining as $\hat{G}$ the restriction of G to the set $P(G) \cup G_b.\mathtt{E}$ (i.e., , $G.\mathtt{E} \setminus D(G)$), it only remains to show that there is no event event of t in $G \setminus \hat{G}$, for some thread $t > \mathtt{tid}(r)$. Let $t = \overset{a}{\underset{rv\ d}{\Rightarrow}}$ be the first revisit step after G in S_b and G'' the execution immediately before t. By definition of $D(\cdot)$, it is $d \in D(G'')$. From [17], revisits happen from right to left, i.e., $\mathtt{tid}(a) > \mathtt{tid}(d)$. As we showed earlier, there is no event $d' \in D(G'')$ such that $d' < r$, and therefore it is $\mathtt{tid}(r) <= \mathtt{tid}(d) < \mathtt{tid}(a)$. Finally, no event e of thread $\mathtt{tid}(a)$ in G is in $D(G)$; otherwise e must have been deleted in a later step, but t is the first revisit step after G and from Lemma 3 we get that no later step can delete a. □