Formal Methods in Computer-Aided Design 2023

Optimal Bounded Partial Order Reduction

Iason Marmanis
Max Planck Institute for Software Systems
Kaiserslautern, Germany
imarmanis@mpi-sws.org

Viktor Vafeiadis
Max Planck Institute for Software Systems
Kaiserslautern, Germany
viktor@mpi-sws.org

***Abstract*—Preemption bounding (PB) and dynamic partial order reduction (DPOR) are two key techniques for scaling up the model checking of concurrent software. Attempts to combine them have so far been suboptimal: they either explore redundant executions (that DPOR alone would eliminate) or executions exceeding the desired bound (that PB alone would not consider).**

By bounding the number of rounds of a round-robin scheduler instead of the number of preemptions, we obtain the first optimal bounded partial order reduction algorithm. Our approach has two additional benefits: (1) it makes checking boundedness of a Mazurkiewicz trace linear-time (instead of NP-hard) and (2) it extends smoothly to weak memory models.

I. Introduction

Even under *sequential consistency* (SC) [18], to make automated verification of concurrent programs feasible, one typically has to restrict their state space in several unsound ways, such as considering only executions with up to K recursive calls, L loop iterations, N concurrent threads, and even M preemptive context switches between them. In this paper, we will focus on the latter restriction, which is known as *preemption bounding* (PB) or *context bounding* [23].

Bounding such quantities is generally sufficient for finding safety errors in programs and can provide reasonable confidence in the correctness of programs whose full verification is intractable. PB is especially good in that regard: it achieves great state-space reduction (since the number of executions of a concurrent program is exponential in the number of preemptions) and bug coverage (because bugs in practice can be exposed with a very small number of preemptions [22]).

Bounding, however, often destroys symmetries in a program, which lessens the effect of sound state-space reduction techniques. In particular, PB does not work well with *dynamic partial order reduction* (DPOR) [9], which calls two executions of a concurrent program equivalent if they differ only in the order of commuting operations (e.g., two accesses to different shared memory locations) and strives to explore only one execution per equivalence class.

Combining DPOR and bounding optimally is non-trivial. Coons et al. [8] weaken the benefit of DPOR leading to the (redundant) exploration of multiple equivalent interleavings; whereas Marmanis et al. [19] weaken the benefit of PB and often require the exploration of executions with more preemptions than the desired bound. Moreover, both approaches suffer from the NP-hardness of checking whether a given execution is equivalent to some execution with at most M preemptions.

In this paper, we provide an optimal combination of these two techniques by changing the bounded quantity. Rather than assuming a completely non-deterministic scheduler and bounding the number of preemptive context switches, we assume the presence of a round-robin scheduler under a fixed ordering of the threads (e.g., in increasing thread-identifier order) and bound the number of rounds such a scheduler can take. This change has two immediate consequences:

1) Checking whether an execution is below the desired bound can be decided in linear time (see §II).
2) The optimal DPOR exploration procedure of Kokologiannakis et al. [12] is monotone in the number of scheduling rounds (see §III).

Therefore, by stopping the exploration of any execution prefix that exceeds the desired bound, we immediately obtain a sound, complete, and optimal bounded partial order reduction algorithm called Rounder (see §IV), which enables the bounded verification of programs whose unbounded verification is intractable (see §VI). Our optimal bounding approach extends seamlessly to weak memory models for bounding metrics that are similar to the number of scheduling rounds or that constrain only the non-SC part of executions (see §V).

II. Bounding the Number of Scheduling Rounds

A. Program Traces and Execution Graphs

A program trace τ is a sequence of *events*, each corresponding to the execution of a single thread instruction, such as a *read* (R) or a *write* (W) of a certain location and value. In a sequentially consistent trace, every read event r in the trace reads the value written by the last write event in the same location that appears before r in the trace. Two traces are *(Mazurkiewicz-)equivalent* if they only differ in the order of commuting instructions (of different threads) [20].

Instead of using traces, several recent DPOR algorithms [11, 12, 14] directly explore their equivalence classes, succinctly represented as *execution graphs*. An execution (graph) G consists of a set of events $G.\mathtt{E}$ including one initialization write event for each memory location and the following set of directed edges that reflect the ordering between the events:

- the *program order* $G.\mathtt{po}$, which orders events of the same thread in their control-flow order and initialization write events before all non-initialization events,
- the *coherence order* $G.\mathtt{co}$, which (totally) orders same-location write events, and

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_16

This article is licensed under a Creative Commons Attribution 4.0 International License

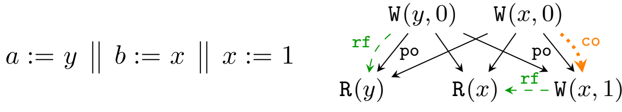

Fig. 1. A program (left) and one of its execution graphs (right).

- the *reads-from* $G.\mathtt{rf}$, which orders every read event after the write event that it reads from.

We write $G|_S$ for the restriction of execution G to the set of events S; and we say that G is a prefix of G' if $G = G'|_{G.\mathtt{E}}$.

In Fig. 1, we show a program with three threads and one of its execution graphs. The graph contains two initialization writes (one for x and one for y along with three events corresponding to the memory accesses of the program.

Given two relations X and Y, we define their composition $X;Y \triangleq \{\langle x,z\rangle \mid \exists y.\, \langle x,y\rangle \in X \wedge \langle y,z\rangle \in Y\}$ and the inverse $X^{-1} \triangleq \{\langle x,y\rangle \mid \langle y,x\rangle \in X\}$ of X. Finally, we write X^+ for the transitive closure of X.

We now define the following four derived relations:

- The *causality order*, $G.\mathtt{porf} \triangleq (G.\mathtt{po} \cup G.\mathtt{rf})^+$, captures dependencies between events due to the program order and the reads-from relation: an event b causally ordered after an event a cannot be executed before a because it depends on a's execution.
- The *from-reads* relation, $G.\mathtt{fr} \triangleq G.\mathtt{rf}^{-1}; G.\mathtt{co}$, orders every read event before the same-location writes that are co-after than the write that the read it is reading from.
- The *extended coherence order*, $G.\mathtt{eco} \triangleq (G.\mathtt{rf} \cup G.\mathtt{co} \cup G.\mathtt{fr})^+$, orders all same-location access events apart from two reads that read from the same write—their order is immaterial.
- The *SC order*, $G.\mathtt{sc} \triangleq (G.\mathtt{po} \cup G.\mathtt{eco})^+$, puts together orderings due to the program and due to coherence.

An execution G is *(sequentially) consistent* if $G.\mathtt{sc}$ is irreflexive. A consistent execution G represents the equivalence class of the set of all linearizations of $G.\mathtt{sc}$.

Other memory models require only certain subsets of the $G.\mathtt{sc}$ relation to be irreflexive. For example, coherence requires $G.\mathtt{po}; G.\mathtt{eco}$ to be irreflexive, while release-acquire consistency requires $G.\mathtt{porf}; G.\mathtt{eco}$ to be irreflexive.

B. Round-Robin Rounds

We define the (round-robin) *rounds* of a program trace $\tau = e_1, e_2, \ldots, e_k$ to be the number of times a round-robin scheduler needs to start again from the first thread to generate the trace τ, i.e., $\mathsf{rounds}(\tau) \triangleq |\{e_i \mid \mathtt{tid}(e_i) > \mathtt{tid}(e_{i+1})\}|$. where $\mathtt{tid}(e)$ returns the thread identifier of event e.

The notion of rounds can be naturally lifted to execution graphs: the rounds of an execution graph is the least among the rounds of the traces it represents, i.e., $\mathsf{rounds}(G) \triangleq \min\{\mathsf{rounds}(\tau) \mid \tau \text{ linearizes } G.\mathtt{sc}\}$.

The execution graph in Fig. 1 represents the following three traces: (a) $a := y\,; x := 1\,; b := x$, (b) $x := 1\,; a := y\,; b := x$, and (c) $x := 1\,; b := x\,; a := y$. It has one round, since traces (a) and (b) have one round, whereas trace (c) has two rounds.

Algorithm 1 Greedy algorithm for $\mathsf{rounds}(G)$

```
1: procedure rounds(G)
2:   S ← G.E
3:   rounds ← 0
4:   while S ≠ ∅ do
5:     rounds ← rounds + 1
6:     for i ← 1 .. N do
7:       while ( ∃e ∈ S. tid(e) = i
                 ∧ ∄e' ∈ S. ⟨e', e⟩ ∈ G.sc ) do
8:         S ← S \ {e}
9:   return rounds − 1
```

Clearly, to compute the rounds of an execution G we do not have to enumerate the traces of G. Assuming G has N threads, Algorithm 1 computes $\mathsf{rounds}(G)$ with a greedy approach: it follows the scheduling of the round-robin scheduler adding as many events from the current thread as possible. Any trace τ' has at least as many rounds as the trace τ that $\mathsf{rounds}(.)$ (implicitly) constructs. To see this, observe that at the first point where τ and τ' differ, τ' could be extended with the event e of thread t, but instead moved to the next thread and possibly incurred an additional round.

We note that $\mathsf{rounds}(.)$ is *monotone* w.r.t. the prefix relation.

Proposition 1. *Given two consistent executions G and G', if G is a prefix of G', then $\mathsf{rounds}(G) \leq \mathsf{rounds}(G')$.*

Proof. Consider a trace τ' of G' with $\mathsf{rounds}(G')$ rounds. Restricting it to the events of G yields a trace τ of $G.\mathtt{E}$ with at most $\mathsf{rounds}(G')$ rounds. □

C. Rounds versus Context Switches

We say that a program trace incurs a *context switch* whenever adjacent elements of the trace belong to different threads. A *preemptive* context switch between two events is one where the thread of the first event could have continued execution: it is neither blocked nor finished. As with scheduling rounds, we straightforwardly count the number of (preemptive) context switches in a trace and lift that definition to execution graphs.

Although the formal definitions of scheduling rounds and context switches are very similar, the number of scheduling rounds and the number of context switches of a particular trace can differ widely. The reason is that one scheduling round in a program with N threads can contain events from at least one and at most N threads. Consequently, a round-robin execution of a program with N threads and K rounds can have at most $K \times N$ context switches.

Conversely, an arbitrary execution with N threads and C context-switches can be generated by a round-robin scheduler with at most $C - \lfloor C/N \rfloor$ rounds. To see this, consider the worst-case scenario where as many of the context-switches as possible incur a new round: at least $\lfloor C/N \rfloor$ of them originate from the same thread and therefore at least $\lfloor C/N \rfloor$ increase the thread identifier of the current thread, which does not incur a new round.

III. Optimal Unbounded DPOR

In this section, we recall the TruSt algorithm [12] in Algorithm 2. With every execution graph G, TruSt keeps track of a total order $<_G$ on G.E, which corresponds to the order they were added to the graph.

Algorithm 2 TruSt's exploration algorithm

```
1:  procedure VERIFY(P)
2:     VISIT_P(G_∅)
3:  procedure VISIT_P(G)
4:     if ¬consistent(G) then return
5:     switch a ← next_P(G) do
6:        case a = ⊥
7:           return "Visited full execution graph G"
8:        case a ∈ error
9:           return "Visited erroneous execution G"
10:       case a ∈ R
11:          for w ∈ G.W_loc(a) do VISIT_P(SetRF(G, a, w))
12:       case a ∈ W
13:          VISITCOS_P(G, a)
14:          for r ∈ G.R_loc(a) s.t. ⟨r, a⟩ ∉ G.porf do
15:             D ← {e ∈ G.E | r <_G e ∧ ⟨e, a⟩ ∉ G.porf}
16:             if ISMAXIMAL(G, {r} ∪ D, a) then
17:                VISITCOS_P(SetRF(G|_{G.E\D}, r, a), a)
18:       case _
19:          VISIT_P(G)
20: procedure VISITCOS_P(G, a)
21:    for w_p ∈ G.W_loc(a) do VISIT_P(SetCO(G, w_p, a))
```

TruSt's exploration of the execution of program P starts by invoking Visit_P on the empty execution graph. At each step, TruSt checks that the current execution graph is inconsistent and drops it if so (line 4). Otherwise, TruSt augments the current execution with the next event a picked by the scheduler (line 5), and proceeds differently depending on the type of a. The interesting cases are when a is a read or a write.

In the case of a read event, TruSt considers all the executions where a reads from some write event w in the same location as a. ($\mathsf{SetRF}(G, a, w)$ modifies G so that a reads from w.)

In the case of a write event a, TruSt first considers every possible co-placement for a (placing it directly after each write w_p to the same location as a via $\mathsf{SetCO}(G, w_p, a)$, line 21). Second, it considers *revisiting* each previously added read r of the same location as a that does not causally precede a (line 14). The revisit operation removes from the execution all events added after r that do not causally precede a (line 17).

To perform the revisit, TruSt checks a *maximality* condition for the set of events that would be removed from the execution graph (line 16). Intuitively, $\text{IsMaximal}(G, S, a)$ checks if it is possible to reconstruct G from the restriction of G to the events of $G.\mathtt{E} \setminus S$, by adding the events of S one by one in the order prescribed by $<_G$ in a *coherence-maximal* way: each read event must read from the co-maximal same-location write, and each write must be added at the end of co. This condition is necessary to guarantee *optimality*, i.e., no execution graph is explored twice. We omit the concrete definition of IsMaximal and refer the reader to Kokologiannakis et al. [12] for details about this condition.

Assuming that $\mathsf{next}_P(.)$ always picks an event from the leftmost available thread, we can prove that the steps of TruSt are monotone w.r.t. to the rounds function, i.e., if $\text{Visit}_P(G)$ invokes $\text{Visit}_P(G')$, then $\mathsf{rounds}(G) \leq \mathsf{rounds}(G')$.

To prove this monotonicity, we need the following corollary that follows directly from TruSt's proof of correctness [13, Prop A.22 (P4)]:

Corollary 1. *Let G be a consistent execution visited by Algorithm 2 and e be either the revisited read, if the last step was a revisit, or the last event added, otherwise. Then, there is no $G.\mathtt{porf}$-maximal event e' such that $\mathtt{tid}(e') > \mathtt{tid}(e)$.*

Proposition 2. *Assuming that $\mathsf{next}_P(.)$ always picks an event from the leftmost available thread, if a call to $\text{Visit}_P(G)$ directly calls $\text{Visit}_P(G')$, then $\mathsf{rounds}(G) \leq \mathsf{rounds}(G')$.*

Proof. For calls from lines 11 and 13, we immediately have $\mathsf{rounds}(G) \leq \mathsf{rounds}(G')$ from Prop. 1.

The remaining case is when the call to $\text{Visit}(P, G')$ results from a revisit at line 17. Let $\hat{G}$ be the execution that results from G' by removing the added write a and the revisited read r, and S be the linearization of G.po on the events in $G.\mathtt{E} \setminus \hat{G}.\mathtt{E}$ in non-decreasing thread identifier order. Note that r is the first event in S. From TruSt's proof of correctness [13, Prop A.22 (P3)], G can be obtained from $\hat{G}$ by adding the missing events in the order they appear in S in a coherence-maximal way. We now consider two cases, depending on whether there exists a trace τ of $\hat{G}$ with $\mathsf{rounds}(\hat{G})$ rounds such that $\mathtt{tid}(\mathsf{last}(\tau)) \leq \mathtt{tid}(r)$, where $\mathsf{last}(.)$ returns the last event of a trace.

If there is such a trace τ, then it is easy to see that $\tau +\!\!+ S$ is a trace of G and $\mathsf{rounds}(\tau +\!\!+ S) = \mathsf{rounds}(\tau)$. Thus we have $\mathsf{rounds}(G) \leq \mathsf{rounds}(\tau +\!\!+ S) = \mathsf{rounds}(\tau) = \mathsf{rounds}(\hat{G})$. From monotonicity, we have $\mathsf{rounds}(\hat{G}) \leq \mathsf{rounds}(G')$, which gives us the desired $\mathsf{rounds}(G) \leq \mathsf{rounds}(G')$.

Otherwise, let $\hat{\tau}$ be a trace of $\hat{G}$ with $\mathsf{rounds}(\hat{G})$ rounds. Again, $\hat{\tau} +\!\!+ S$ is a trace of G, but $\mathsf{rounds}(\hat{\tau} +\!\!+ S) = \mathsf{rounds}(\hat{\tau}) + 1$, because $\mathtt{tid}(\mathsf{last}(\hat{\tau})) \leq \mathtt{tid}(r)$. Since $\mathsf{rounds}(G) \leq \mathsf{rounds}(\hat{\tau} +\!\!+ S)$ and $\mathsf{rounds}(\hat{\tau}) = \mathsf{rounds}(\hat{G})$, showing that $\mathsf{rounds}(\hat{G}) \leq \mathsf{rounds}(G') - 1$ suffices to prove that $\mathsf{rounds}(G) \leq \mathsf{rounds}(G')$.

Assume the opposite, i.e., $\mathsf{rounds}(\hat{G}) \geq \mathsf{rounds}(G')$ and let $K = \mathsf{rounds}(G')$. From monotonicity, $\mathsf{rounds}(\hat{G}) \leq K$, and thus $\mathsf{rounds}(\hat{G}) = K$. Let G'' be the execution that results from removing r from G'. Since $\hat{G} \sqsubseteq G'' \sqsubseteq G'$, from monotonicity, it is also $\mathsf{rounds}(G'') = K$. From TruSt's proof of correctness [13, Prop A.22 (P9)], because a revisited r in G', $\mathtt{tid}(a) > \mathtt{tid}(r)$. From Corollary 1 for G', any event e' with $\mathtt{tid}(e') > \mathtt{tid}(r)$ is not $G'.\mathtt{porf}$-maximal, and therefore the only event e' in G'' with $\mathtt{tid}(e') > \mathtt{tid}(r)$ that is $G''.\mathtt{porf}$-maximal is the write a. Any trace τ'' with K rounds must end with a, otherwise we can remove a from τ'' and obtain a trace

$\hat{\tau}'$ of $\hat{G}$ with at most K (and therefore exactly K) rounds such that $\mathtt{tid}(\mathsf{last}(\hat{\tau}')) \leq \mathtt{tid}(r)$, which contradicts the hypothesis. Let τ' be a trace of G' with K rounds. Removing r from τ' results in a trace τ'' of G'' with at most K (and therefore exactly K) rounds. Since τ'' ends with a, and r must be after a in τ', it is $\tau' = \tau +\!\!+ [r, a]$, for a trace τ of $\hat{G}$. Therefore $\mathsf{rounds}(\tau') = \mathsf{rounds}(\tau) + 1$ ($\mathtt{tid}(a) > \mathtt{tid}(r)$). This leads to a contradiction: $\mathsf{rounds}(\hat{G}) = \mathsf{rounds}(\tau) = \mathsf{rounds}(\tau') - 1 = \mathsf{rounds}(G') - 1 = \mathsf{rounds}(\hat{G}) - 1$. □

IV. Optimal Bounded DPOR

Given Prop. 2, we can trivially obtain an algorithm that explores all executions of a program *P* with up to k rounds. Let ROUNDER be Algorithm 2 that, apart from consistency, also checks whether $\mathsf{rounds}(G) \leq k$ at line 4.

ROUNDER is sound, complete, and optimal. Soundness is trivial because any execution that is not consistent or exceeds the bound k is dropped. Completeness of ROUNDER, i.e., ROUNDER explores every consistent execution of *P* with up to k rounds, follows from the completeness of TruSt and Prop. 2. Optimality, i.e., no execution graph is explored twice, is inherited from the TruSt algorithm because ROUNDER explores a subset of the executions that TruSt does.

Theorem 1. ROUNDER *is sound, complete, and optimal.*

V. Extensions for Weak Memory Models

While in the previous sections we have focused on sequentially consistent executions, the framework can also be used for weaker memory models, such as x86-TSO, PSO, and RC11. In fact, TruSt is parametric in the choice of the memory model, provided it respects some common assumptions, such as ruling out porf cycles.

Our optimal bounding approach can be similarly generalized to such memory models by choosing a bounding function that validates Prop. 2. A sufficient condition is for the function (a) to be monotone w.r.t. the prefix relation and (b) to not be affected by the coherence maximal addition of an event. To see this, note that any execution graph that TruSt visits in order to reach a final execution graph G_f is a prefix of G_f that is (possibly) extended with coherence-maximally added events [19].

One suitable such function is the modification of Algorithm 1 by changing $G.\mathtt{sc}$ to just $G.\mathtt{porf}$. Another is to count the number of simple sc cycles in an execution graph, or the number of events participating in such cycles. It is also possible to combine the results of multiple such functions by any monotone operation (e.g., addition or to return a tuple).

VI. Evaluation

We implemented ROUNDER on top of the GENMC tool [15], which implements the TruSt algorithm. To evaluate ROUNDER, we investigate (a) how many rounds are usually enough to discover concurrency bugs (§ VI-A), and (b) how efficient is ROUNDER for that number of rounds (§ VI-B).

Our evaluation shows that 2 rounds suffice to uncover almost all concurrency bugs, and for a bound of 2, bounded search is generally much faster that a plain DPOR algorithm.

TABLE I
ROUNDER'S SPEEDUP COMPARED TO GENMC

Benchmark	GENMC Time (s)	$k = 2$ Speedup	$k = 3$ Speedup
bstack(5)	90.37	37.50	5.11
bstack(6)	859.83	104.35	9.89
bstack2(8)	174.72	95.48	9.84
bstack2(9)	730.50	243.50	19.37
dglm-queue(6)	105.58	9.36	2.17
dglm-queue(7)	589.54	22.49	3.88
dglm-oe(7)	22.23	3.01	1.02
dglm-oe(8)	33.33	3.60	1.11
dglm-fifo(7)	24.40	1.81	1.10
dglm-fifo(8)	40.48	1.88	0.83
ms-queue(6)	296.69	42.44	4.96
ms-queue(7)	148.64	34.09	4.61
ms-queue(8)	660.85	81.39	8.50
ms-oe(6)	242.88	23.13	4.26
ms-oe(7)	490.78	34.22	5.62
ttas-lock2(7)	28.13	7.99	2.12
ttas-lock2(8)	149.35	19.20	3.93
ttas-lock3	424.31	22.30	3.80

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 ROUNDER. All reported times are in seconds. We set a timeout limit of 30 minutes.

A. Rounds and Bug Discovery

To evaluate how many rounds are sufficient to discover concurrency bugs in practice, we run ROUNDER on the two sets of benchmarks used in the evaluation of BUSTER [19].

For the first set of benchmarks consisting of programs from SV-COMP [24] and SCTBench [25], ROUNDER discovered all bugs with only two rounds, apart from one benchmark with 100 threads where it times out before discovering the bug. ROUNDER managed to find one more bug than BUSTER before timing out because of the significantly reduced overhead of the bound calculation.

For the second set of benchmarks, consisting of concurrent data structures with induced bugs, ROUNDER discovered again all bugs with two rounds, with the exception of two benchmarks where it timed out, while BUSTER does not. The reason for this is that the number of executions grows faster as the round-robin bound increases compared to when the preemption-bound increases.

B. Bounding Efficiency

To evaluate how efficient ROUNDER is, we compared its execution time for bounds of two and three against the unbounded DPOR algorithm implemented in GENMC. We again used a set of correct concurrent data structures as benchmarks.

Our results are summarized in Table I. We report the execution time of GENMC and the speedup when run with ROUNDER for bounds of two and three. In most benchmarks, ROUNDER is significantly faster under both bound values. For the "dglm-oe" and "dglm-fifo" benchmarks, little to no

speedup is observed because ROUNDER explores most to all program executions for these small bounds.

In comparison to BUSTER, we note that the execution time of ROUNDER grows faster as the bound increases. This happens because scheduling rounds are a coarser-grained bounding metric than preemptions: one additional round typically allows many more executions than one additional preemption.

VII. Related Work and Conclusions

In this paper, we have presented the first optimal bounded DPOR algorithm. While bounding the number of round-robin scheduling rounds in the context of DPOR is a novel contribution of this paper, the bound itself is not new. It was first used by Lal et al. [17] as a technical device to show that preemption-bounded verification of concurrent pushdown automata is decidable, and has since been used for related decidability results.

Two other works have tried to integrate notions of concurrency bounding into DPOR algorithms, albeit nonoptimally. Specifically, Musuvathi et al. [21] developed the BPOR algorithm, which combines DPOR with a preemption-bound search by weakening the reduction obtained by DPOR to avoid exploring any executions with a larger number of preemptions than the desired bound. As a result, their algorithm explores redundant equivalent executions leading to poor performance, which is often worse than the state of the art in unbounded DPOR. More recently, Marmanis et al. [19] developed a different sound approach for combining DPOR and PB by allowing the exploration of executions that exceed by the desired bound by a certain margin equal to the number of threads in the program minus two. Their tool, BUSTER, avoids any redundant exploration, and so is generally faster than unbounded DPOR; it does, however, typically explore a large number executions exceeding the bound, negatively impacting its performance. Both approaches are further limited by the need to determine whether a given execution graph (Mazurkiewicz trace) exceeds the desired preemption bound, which is an NP-complete problem [21].

A large body of work on DPOR devoted on coarser equivalence relations [4, 6, 7, 10, 14] and on supporting weak memory models [1, 3, 11, 14, 16]. These works are mostly orthogonal to our extension over TruSt [12], and can likely be integrated into ROUNDER.

Finally, Abdulla et al. [2] and Atig et al. [5] have proposed bounds for the TSO and Power memory models, but being based on preemption bounding, they are not very suitable for integration with DPOR. In contrast, the proposed bounds of §V, while coarser, can be integrated smoothly into ROUNDER.

Acknowledgments

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

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, 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Akash Lal and Thomas Reps. "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis". In: *CAV 2008*. Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 37–51. ISBN: 978-3-540-70545-1.

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

[19] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. "Reconciling Preemption Bounding with DPOR". In: *TACAS 2023*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 85–104. ISBN: 978-3-031-30823-9.

[20] Antoni Mazurkiewicz. "Trace Theory". In: *PNAROMC 1987*. Vol. 255. LNCS. Berlin, Heidelberg: Springer, 1987, pp. 279–324. DOI: 10.1007/3-540-17906-2_30. URL: http://dx.doi.org/10.1007/3-540-17906-2_30.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.