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 \( \tau \) 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 \( G.E \) 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 \( G.po \), which orders events of the same thread in their control-flow order and initialization write events before all non-initialization events,
- the coherence order \( G.co \), 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
A program (left) and one of its execution graphs (right).

- the reads-from $G.rf$, 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 $G = G'|_E$.

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 $X;Y \triangleq \{(x,z) \mid \exists y. (x,y) \in X \land (y,z) \in Y\}$ and the inverse $X^{-1} \triangleq \{(x,y) \mid (y,x) \in X\}$ of $X$. Finally, we write $X^+$ for the transitive closure of $X$.

We now define the following four derived relations:

- The causality order, $G.po \triangleq (G.po \cup G.rf)^+$, 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, $G.fr \triangleq G.rf^{-1}$; $G.co$, orders every read event before the same-location writes that are co-after than the write that it is reading from.

- The extended coherence order, $G.eco \triangleq (G.rf \cup G.co \cup G.fr)^+$, orders all same-location access events apart from two reads that read from the same write—their order is immaterial.

- The $SC$ order, $G.sc \triangleq (G.po \cup G.eco)^+$, puts together orderings due to the program and due to coherence.

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

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

B. Round-Robin Rounds

We define the (round-robin) rounds of a program trace $\tau = e_1, e_2, \ldots, e_N$ to be the number of times a round-robin scheduler needs to start again from the first thread to generate the trace $\tau$, i.e., rounds($\tau$) $\triangleq \min \{ \left[ \text{tid}(e_j) - \text{tid}(e_{j+1}) \right] \}$. where $\text{tid}(e)$ 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., rounds($G$) $\triangleq \min \{ \text{rounds}(\tau) \mid \tau \text{ linearizes } G.sc \}$.

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 rounds($G$)

1: procedure rounds($G$)
2: $S \leftarrow G.E$
3: rounds $\leftarrow 0$
4: while $S \neq \emptyset$ do
5: rounds $\leftarrow$ rounds + 1
6: for $i \leftarrow 1$ to $N$ do
7: while $\exists e \in S. \text{tid}(e) = i$
8: $S \leftarrow S \setminus \{e\}$
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 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 $\tau'$ has at least as many rounds as the trace $\tau$ that rounds($\tau'$) (implicitly) constructs. To see this, observe that at the first point where $\tau$ and $\tau'$ differ, $\tau'$ 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 rounds($\tau'$) 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 rounds($G$) $\leq$ rounds($G'$).

Proof. Consider a trace $\tau'$ of $G'$ with rounds($G'$) rounds. Restricting it to the events of $G$ yields a trace $\tau$ of $G.E$ with at most 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 $C - \lceil C/N \rceil$ rounds. To see this, consider the worst-case scenario where as many of the context-switches as possible incur a new round: at least $\lceil C/N \rceil$ of them originate from the same thread and therefore at least $\lceil C/N \rceil$ increase the thread identifier of the current thread, which does not incur a new round.
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: \hspace{1em} VISIT$_P(G_{\emptyset})$
3: 3: procedure VISIT$_P(G)$
4: \hspace{1em} if $\neg$-consistent($G$) then return
5: \hspace{1em} switch $a \leftarrow$ next$_P(G)$ do
6: \hspace{1em} \hspace{1em} case $a = \bot$
7: \hspace{1em} \hspace{1em} return “Visited full execution graph $G$”
8: \hspace{1em} \hspace{1em} case $a \in$ error
9: \hspace{1em} \hspace{1em} return “Visited erroneous execution $G$”
10: \hspace{1em} case $a \in R$
11: \hspace{1em} \hspace{1em} for $w \in G.W_{loc(a)}$ do VISIT$_P(SetRF(G, a, w))$
12: \hspace{1em} case $a \in \bar{W}$$
13: \hspace{1em} \hspace{1em} VISITCOS$_P(G, a)$
14: \hspace{1em} \hspace{1em} for $r \in G.B_{loc(a)}$ s.t. $(r, a) \notin G.\text{porf}$ do
15: \hspace{1em} \hspace{1em} D $\leftarrow \{ e \in G.E \mid r <_G e \land (e, a) \notin G.\text{porf} \}$
16: \hspace{1em} \hspace{1em} if ISMAXIMAL($G, \{r\} \cup D, a$) then
17: \hspace{1em} \hspace{1em} \hspace{1em} VISITCOS$_P(SetRF(G|_{G.E\setminus D}, r, a), a)$
18: \hspace{1em} \hspace{1em} \hspace{1em} VISIT$_P(G)$
19: \hspace{1em} \hspace{1em} case __
20: \hspace{1em} \hspace{1em} VISIT$_P(G)$
21: \hspace{1em} \hspace{1em} for $w_p \in G.W_{loc(a)}$ 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 $a$. (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 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, ISMAXIMAL($G, S, a$) checks if it is possible to reconstruct $G$ from the restriction of $G$ to the events of $G.E\setminus S$, 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 next$_P(\cdot)$ 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 VISIT$_P(G)$ invokes VISIT$_P(G')$, then rounds($G$) $\leq$ 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 $G.\text{porf}$-maximal event $e'$ such that $\text{tid}(e') > \text{tid}(e)$.

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

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

The remaining case is when the call to VISIT($P, G'$) results from a revisit at line 17. Let $G$ 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 $G.E\setminus G.E$ 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 $G$ 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 $\tau$ of $G$ with rounds($\tau$) rounds such that $\text{tid}$(last(\tau)) $\leq$ $\text{tid}$(r), where last(\cdot) returns the last event of a trace.

If there is such a trace $\tau$, then it is easy to see that $\tau + S$ is a trace of $G$ and rounds($\tau + S$) = rounds($\tau$). Thus we have rounds($G$) $\leq$ rounds($\tau + S$) = rounds($\tau$) = rounds($G$).

From monotonicity, we have rounds($G$) $\leq$ rounds($G'$), which gives us the desired rounds($G$) $\leq$ rounds($G'$).

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

Assume the opposite, i.e., rounds($G$) $\geq$ rounds($G'$) and let $K = \text{rounds}(G')$. From monotonicity, rounds($G$) $\leq$ $K$, and thus rounds($G$) = $K$. Let $G''$ be the execution that results from removing $r$ from $G'$. Since $G \subseteq G'' \subseteq G'$, from monotonicity, it is also rounds($G''$) = $K$. From TruSt’s proof of correctness [13, Prop A.22 (P9)], because a revisited read $r$ in $G'$, $\text{tid}(a) > \text{tid}(r)$. From Corollary 1 for $G'$, any event $e'$ with $\text{tid}(e')$ $>$ $\text{tid}(r)$ is not $G'.\text{porf}$-maximal, and therefore the only event $e'$ in $G''$ with $\text{tid}(e')$ $>$ $\text{tid}(r)$ that is $G''.\text{porf}$-maximal is the write $a$. Any trace $\tau''$ with $K$ rounds must end with $a$, otherwise we can remove $a$ from $\tau''$ and obtain a trace.
\( \hat{\tau}' \) of \( \hat{G} \) with at most \( K \) (and therefore exactly \( K \)) rounds such that \( \text{tid}(\text{last}(\hat{\tau}')) \leq \text{tid}(r) \), which contradicts the hypothesis. Let \( \tau' \) be a trace of \( G' \) with \( K \) rounds. Removing \( r \) from \( \tau' \) results in a trace \( \tau'' \) of \( G'' \) with at most \( K \) (and therefore exactly \( K \)) rounds. Since \( \tau'' \) ends with \( a \), and \( r \) must be after \( a \) in \( \tau' \), it is \( \tau' = \tau' + [r, a] \), for a trace \( \tau \) of \( G \). Therefore \( \text{rounds}(\tau') = \text{rounds}(\tau) + 1 \) \((\text{tid}(a) > \text{tid}(r))\). This leads to a contradiction: \( \text{rounds}(\hat{G}) = \text{rounds}(\tau) = \text{rounds}(\tau') - 1 = \text{rounds}(G') - 1 = \text{rounds}(\hat{G}) - 1 \). \( \square \)

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 \textsc{Rounder} be Algorithm 2 that, apart from consistency, also checks whether \( \text{rounds}(G) \leq k \) at line 4.

\textsc{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 \textsc{Rounder}, i.e., \textsc{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 \textsc{Rounder} explores a subset of the executions that TruSt does.

Theorem 1. \textsc{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 \textit{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 \( G, \text{sC} \) to just \( G, \text{porf} \). Another is to count the number of simple \textit{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 \textsc{Rounder} on top of the \textsc{GenMC} tool [15], which implements the TruSt algorithm. To evaluate \textsc{Rounder}, we investigate (a) how many rounds are usually enough to discover concurrency bugs (§VI-A), and (b) how efficient is \textsc{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>
<thead>
<tr>
<th>Benchmark</th>
<th>GenMC</th>
<th>( k = 2 ) Speedup</th>
<th>( k = 3 ) Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>bstack(5)</td>
<td>90.37</td>
<td>37.50</td>
<td>5.11</td>
</tr>
<tr>
<td>bstack(6)</td>
<td>859.83</td>
<td>104.35</td>
<td>9.89</td>
</tr>
<tr>
<td>bstack2(8)</td>
<td>174.72</td>
<td>95.48</td>
<td>9.84</td>
</tr>
<tr>
<td>bstack2(9)</td>
<td>739.50</td>
<td>243.50</td>
<td>19.37</td>
</tr>
<tr>
<td>dglm-queue(6)</td>
<td>105.58</td>
<td>9.36</td>
<td>2.17</td>
</tr>
<tr>
<td>dglm-queue(7)</td>
<td>589.54</td>
<td>22.49</td>
<td>3.88</td>
</tr>
<tr>
<td>dglm-oe(7)</td>
<td>22.23</td>
<td>3.01</td>
<td>1.02</td>
</tr>
<tr>
<td>dglm-oe(8)</td>
<td>33.33</td>
<td>3.60</td>
<td>1.11</td>
</tr>
<tr>
<td>dglm-fifo(7)</td>
<td>24.40</td>
<td>1.81</td>
<td>1.10</td>
</tr>
<tr>
<td>dglm-fifo(8)</td>
<td>40.48</td>
<td>1.88</td>
<td>0.83</td>
</tr>
<tr>
<td>ms-queue(6)</td>
<td>296.69</td>
<td>42.44</td>
<td>4.96</td>
</tr>
<tr>
<td>ms-queue(7)</td>
<td>148.64</td>
<td>34.09</td>
<td>4.61</td>
</tr>
<tr>
<td>ms-queue(8)</td>
<td>660.85</td>
<td>81.39</td>
<td>8.50</td>
</tr>
<tr>
<td>ms-oe(6)</td>
<td>242.88</td>
<td>23.13</td>
<td>4.26</td>
</tr>
<tr>
<td>ms-oe(7)</td>
<td>490.78</td>
<td>34.22</td>
<td>5.62</td>
</tr>
<tr>
<td>taslock2(6)</td>
<td>28.13</td>
<td>7.99</td>
<td>2.12</td>
</tr>
<tr>
<td>taslock2(7)</td>
<td>149.35</td>
<td>19.20</td>
<td>3.93</td>
</tr>
<tr>
<td>taslock3</td>
<td>424.31</td>
<td>22.30</td>
<td>3.80</td>
</tr>
</tbody>
</table>

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 \textsc{GenMC} and \textsc{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 \textsc{Rounder} on the two sets of benchmarks used in the evaluation of \textsc{Buster} [19].

For the first set of benchmarks consisting of programs from \textsc{SV-COMP} [24] and \textsc{SCTBench} [25], \textsc{Rounder} discovered all bugs with only two rounds, apart from one benchmark with 100 threads where it times out before discovering the bug. \textsc{Rounder} managed to find one more bug than \textsc{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, \textsc{Rounder} discovered again all bugs with two rounds, with the exception of two benchmarks where it timed out, while \textsc{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 \textsc{Rounder} is, we compared its execution time for bounds of two and three against the unbounded DPOR algorithm implemented in \textsc{GenMC}. We again used a set of correct concurrent data structures as benchmarks.

Our results are summarized in Table 1. We report the execution time of \textsc{GenMC} and the speedup when run with \textsc{Rounder} for bounds of two and three. In most benchmarks, \textsc{Rounder} is significantly faster under both bound values. For the “\texttt{dglm-oe}” and “\texttt{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


