Unblocking Dynamic Partial Order Reduction

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis

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

Abstract. Existing dynamic partial order reduction (DPOR) algorithms scale poorly on concurrent data structure benchmarks because they visit a huge number of blocked executions due to spinloops.
In response, we develop AWAMOCHE, a sound, complete, and strongly optimal DPOR algorithm that avoids exploring any useless blocked executions in programs with await and confirmation-CAS loops. Consequently, it outperforms the state-of-the-art, often by an exponential factor.

1 Introduction

Dynamic partial order reduction (DPOR) [13] has been promoted as an effective verification technique for concurrent programs: starting from a single execution of the program under test, DPOR repeatedly reverses the order of conflicting accesses in order to generate all (meaningfully) different program executions.

Applying DPOR in practice, however, reveals a major performance and scalability bottleneck: it explores a huge number of *blocked executions*, often outnumbering the complete program executions by an exponential factor. Blocked executions most commonly occur in programs with *spinloops*, i.e., loops that do not make progress unless some condition holds. Such loops are usually transformed into *assume statements* [18, 14], effectively requiring that the loop exits at its first iteration (and blocking otherwise).

We distinguish three classes of such blocked executions.

The first class occurs in programs with non-terminating spinloops, such as a program awaiting for $x > 42$ in a context where $x = 0$. For this program, modeled as the statement $\texttt{assume}(x > 42)$, DPOR obviously explores a blocked execution as the only existing value for x violates the assume condition. Such blocked executions *should* be explored because they indicate program errors.

The second class occurs in programs with await loops. To see how such loops lead to blocked executions, consider the following program under *sequential consistency* (SC) [23] (initially $x{=}y{=}0$),

$$\begin{array}{l} x := 2 \\ \texttt{assume}(y \leq 1) \end{array} \;\Bigg\|\; \begin{array}{l} y := 2 \\ \texttt{assume}(x \leq 1) \\ y := 1 \end{array}$$

where each `assume` models an await loop, e.g., `do` $a := y$ `while` $(a > 1)$ for the `assume` of the first thread. Suppose that DPOR executes this program in a left-to-right manner, thereby generating the interleaving $x := 2, \texttt{assume}(y \leq 1), y := 2$. At this point, $\texttt{assume}(x \leq 1)$ cannot be executed, since x would read 2. Yet, DPOR cannot simply abort the exploration. To generate the interleaving where the first thread reads $y = 1$, DPOR must consider the case where the read of x is executed before the $x := 2$ assignment. In other words, DPOR has to explore blocked executions in order to generate non-blocked ones.

The third class occurs in programs with confirmation-CAS loops such as:

```
do
  a := x
  b := f(a)
while (¬CAS(x, a, b))
```

which is modeled as:

```
a := x
b := f(a)
assume(CAS(x, a, b))
```

Consider a program comprising two threads running the code above, with a and b being local variables. Suppose that DPOR first obtains the (blocked) trace where both threads concurrently try to perform their CAS: $a_1 := x$, $a_2 := x$, $\texttt{CAS}(x, a_1, b_1)$, $\texttt{CAS}(x, a_2, b_2)$. Trying to satisfy the blocked assume of thread 2 by reversing the CAS instructions is fruitless because then thread 1 will be blocked.

In this paper, we show that exploring blocked executions of the second and third classes is unnecessary.

We develop Awamoche, a sound, complete, and optimal DPOR algorithm that avoids generating any blocked executions for programs with await and confirmation-CAS loops. Our algorithm is *strongly optimal* in that no exploration is wasted: it either yields a complete execution or a termination violation. Awamoche extends TruSt [15], an optimal DPOR algorithm that supports weak memory models and has polynomial space requirements, with three new ideas:

1. Awamoche identifies certain reads as *stale*, meaning that they will never be affected by a race reversal due to TruSt's maximality condition on reversals, and avoids exploring any executions that block on stale-read values.
2. To deal with await loops, since it cannot completely avoid generating executions with blocking reads, Awamoche *revisits* such executions *in place* if a same-location write is later encountered. If no such write is found, then the blocked execution witnesses a program termination bug [21, 25].
3. To effectively deal with confirmation-CAS loops, Awamoche only considers executions where the confirmation succeeds, by reversing not only races between conflicting instructions, but also speculatively revisiting traces with two reads reading from the same write event to enable a later in-place revisit.

As we shall see in §5, supporting these DPOR modifications is by no means trivial when it comes to proving correctness and (strong) optimality. Indeed, TruSt's correctness proof proceeds in a backward manner, assuming a way to determine the last event that was added to a given trace. The presence of in-place and speculative revisits, however, makes this impossible.

We therefore develop a completely different proof that works in a forward manner: from each configuration that is a prefix of a complete trace, we construct

a sequence of steps that will lead to a larger configuration that is also a prefix of the trace. Our proof assumes that same-location writes are causally ordered, which invariably holds in correct data structure benchmarks, but is otherwise more general than TruSt's assuming less about the underlying memory model.

Our contributions can be summarized as follows:

§2 We describe how and why DPOR encounters blocked executions.
§3 We intuitively present AWAMOCHE's three novel key ideas: stale reads, in-place revisits, and speculative revisits.
§4 We describe our algorithm in detail in a memory-model-agnostic framework.
§5 We generalize TruSt's proof and prove AWAMOCHE sound, complete, and strongly optimal.
§6 We evaluate AWAMOCHE, and demonstrate that it outperforms the state-of-the-art, often by an exponential factor.

2 DPOR and Blocked Executions

Before presenting AWAMOCHE, we recall the fundamentals of DPOR (§ 2.1), and explain why spinloops lead to blocked explorations (§ 2.2).

2.1 Dynamic Partial Order Reduction

DPOR algorithms verify a concurrent program by enumerating a representative subset of its interleavings. Specifically, they partition the interleavings into equivalence classes (two interleavings are equivalent if one can be obtained from the other by reordering independent instructions), and strive to explore one interleaving per equivalence class. Optimal algorithms [2, 15] achieve this goal.

DPOR algorithms explore interleavings dynamically. After running the program and obtaining an initial interleaving, they detect *racy* instructions (i.e., instructions accessing the same variable with at least one of them being a write), and proceed to explore an interleaving where the race is reversed.

Let us clarify the exploration procedure with the following example, where both variables x and y are initialized to zero.

$$\begin{array}{r||l} \texttt{if}\ (x = 0) & x := 1 \\ y := 1 & x := 2 \end{array} \qquad (\text{RW+WW})$$

The RW+WW program has 5 interleavings that can be partitioned into 3 equivalence classes. Intuitively, the $y := 1$ is irrelevant because the program contains no other access to y; all that matters is the ordering among the x accesses.

The exploration steps for RW+WW can be seen in Fig. 1[1]. DPOR obtains a full trace of the program, while also recording the transitions that it took at each step at the respective transition's backtrack set (traces ⓪ to ②). After obtaining

[1] The exploration procedure has been simplified for presentational purposes. For a full treatment, please refer to [15, 2].

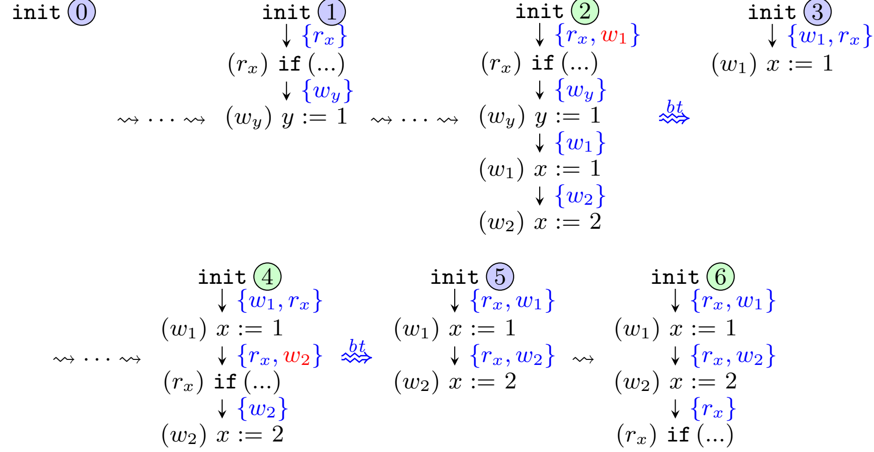

Fig. 1. A DPOR exploration of RW+WW

a full trace, it initiates a race-detection phase. During this phase, DPOR detects the races between r_x and the two writes w_1 and w_2. (While w_1 and w_2 also write the same variable, they do not constitute a race, as they are causally related.) For the first race, DPOR adds w_1 in the backtrack set of the first transition, so that it can subsequently execute w_1 instead of r_x. For the second one, while w_2 is not in the backtrack set of the first transition, w_2 cannot be directly executed as the first transition without its causal predecessors (i.e., w_1) having already executed. Since w_1 is already in the backtrack set of the first transition, DPOR cannot do anything else, and the race-detection phase is over.

After the race-detection phase is complete, the exploration proceeds in an analogous manner: DPOR backtracks to the first transition, fires w_1 instead of r_x (trace ③), re-runs the program to obtain a full trace (trace ④), and initiates another race-detection phase. During the latter, a race between r_x and w_2 is detected, and w_2 is inserted in the backtrack set of the second transition.

Finally, DPOR backtracks to the second transition, executes w_2 instead of r_x (trace ⑤), and eventually obtains the full trace ⑥. During the last race-detection phase of the exploration, DPOR detects the races between r_x and the two writes w_1 and w_2. As r_x is already in the backtrack set of the first two transitions, DPOR has nothing else to do, and thus concludes the exploration.

Observe that DPOR explored one representative trace from each equivalence class (traces ②, ④, and ⑥). To avoid generating multiple equivalent interleavings, optimal DPOR algorithms extend the description above by restricting when a race reversal is considered. In particular, the TruSt algorithm [15] imposes a maximality condition on the part of the trace that is affected by the reversal.

`while` $(x{=}0)$ {} $y := 1$	$\|\|$	$x := 1$ $x := 2$	(RW+WW-L)	`assume`$(x{\neq}0)$ $y := 1$	$\|\|$	$x := 1$ $x := 2$ (RW+WW-A)

Fig. 2. A variation of RW+WW with an await loop (left) and an `assume` (right)

2.2 Assume Statements and DPOR

To see how `assume` statements arise in concurrent programs, suppose that we replace the `if`-statement of RW+WW with an await loop (Fig. 2). Although the change does not really affect the possible outcomes for x, it makes DPOR diverge: DPOR examines executions where the loop terminates in 1, 2, 3, ... steps. Since, however, the loop has no side-effects, we can actually transform it into an `assume`(x) statement, effectively modeling a loop bound of one.

Doing so guarantees DPOR's termination but not its good performance. The reason is ascribed to the very nature of DPOR. Indeed, suppose that DPOR executes the first instruction of the left thread and then blocks due to `assume` statement. At this point, DPOR cannot simply stop the exploration due to the `assume` statement not being satisfied; it has to explore the rest of the program, so that the race reversals make the `assume` succeed. All in all, DPOR explores 2 complete and 1 blocked traces for RW+WW-A.

In general, DPOR cannot know whether some future reversal will ever make an `assume` succeed. Worse yet, it might be the case that there is an exponential number of traces to be explored (due to the other program threads), until DPOR is certain that the `assume` statement cannot be unblocked.

To see this, consider the following program where RW+WW-A runs in parallel with some threads accessing z:

$$\text{RW+WW-A} \parallel z := 1 \parallel a_1 := z \parallel \ldots \parallel a_N := z \qquad \text{(RW+WW-A-PAR)}$$

For the trace of RW+WW-A where the `assume` fails, DPOR fruitlessly explores 2^N traces in the hope that an access to x is found that will unblock the `assume` statement.

Given that executing an `assume` statement that fails leads to blocked executions, one might be tempted to consider a solution where `assume` statements are only scheduled if they succeed. Even though such a solution would eliminate blocking for RW+WW-A, it is not a panacea. To see why, consider a variation of RW+WW-A where the first thread executes `assume`$(x = 0)$ instead of `assume`$(x \neq 0)$. In such a case, the `assume` can be scheduled first (as it succeeds), but reversing the races among the x accesses will lead to blocked executions. It becomes evident that a more sophisticated solution is required.

3 Key Ideas

Awamoche, our optimal DPOR algorithm, extends TruSt [15] with three novel key ideas: stale-read annotations (§ 3.1), in-place revisits (§ 3.2) and speculative

revisits (§ 3.3). As we will shortly see, these ideas guarantee that AWAMOCHE is *strongly optimal*: it never initiates fruitless explorations, and all explorations lead to executions that are either complete or denote termination violations. In the rest of the paper, we call such executions *useful*.

3.1 Avoiding Blocking due to Stale Reads

Race reversals are at the heart of any DPOR algorithm. TruSt distinguishes two categories of race reversals: (1) write-read and write-write reversals, (2) read-write reversals. While the former category can be performed by modifying the trace directly in place (called a "forward revisit"), the latter may require removing events from the trace (called a "backward revisit"). To ensure optimality for backward revisits, TruSt checks a certain maximality condition for the events affected by them, namely the read, which will be reading from a different write, and all events to be deleted.

An immediate consequence is that any read events not satisfying TruSt's maximality condition, which we call *stale reads*, will never be affected by a subsequent revisit. As an example, consider the following program with a read that blocks if it reads 0:

$$x := 1 \parallel \texttt{assume}(x = 1) \qquad \text{(W+R)}$$

After obtaining the trace $x := 1; \texttt{assume}(x = 1)$, TruSt forward-revisits the read in-place, and makes it read 0. At this point, we know that (1) the assume will fail, and (2) that both the read and the events added before it cannot be backward-revisited, due to the read reading non-maximally (which violates TruSt's maximality condition). As such, no useful execution is ever going to be reached, and there is no point in continuing the exploration.

Leveraging the above insight, we make AWAMOCHE immediately drop traces where some assume is not satisfied due to a stale read. To do this, AWAMOCHE automatically annotates reads followed by assume statements with the condition required to satisfy the assume, and discards all forward revisits that do not satisfy the annotation.

Even though stale-read annotations are greatly beneficial in reducing blocking, they are merely a remedy, not a cure. As already mentioned, they are only leveraged in write-read reversals, and are thus sensitive to DPOR's exploration order. To completely eliminate blocking, AWAMOCHE performs in-place and speculative revisits, described in the next sections.

3.2 Handling Await Loops with In-Place Revisits

AWAMOCHE's solution to eliminate blocking is to not blindly reverse all races whenever a trace is blocked, but rather to only try and reverse those that might unblock the exploration.

As an example, consider RW+WW-A-PAR (Fig. 3). After AWAMOCHE obtains the first full trace, it detects the races among the z accesses, as well as the $\langle r_x, w_1 \rangle$

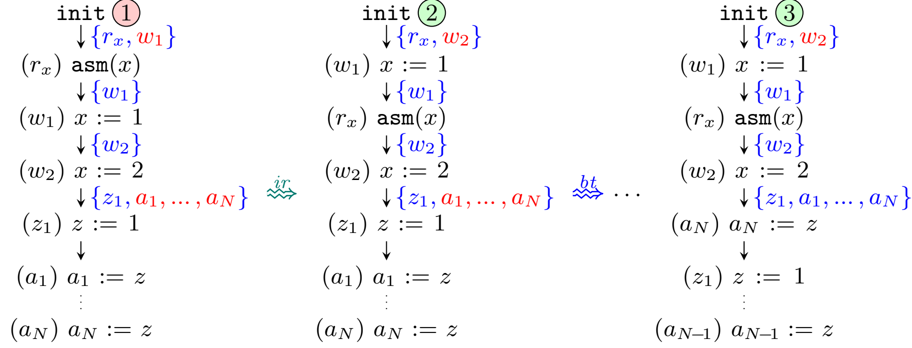

Fig. 3. Key steps in AWAMOCHE's exploration of RW+WW-A-PAR

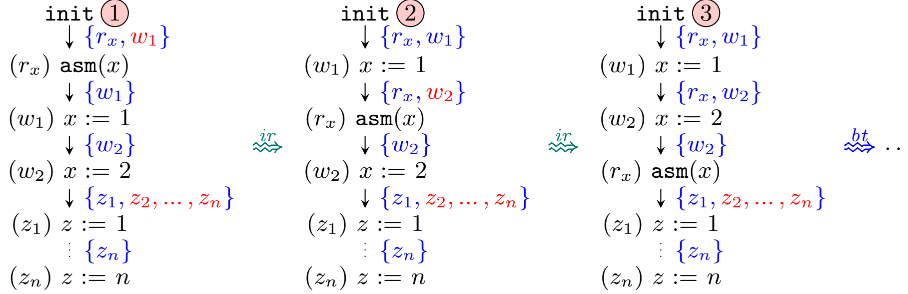

Fig. 4. An AWAMOCHE exploration of RW+WW

race. (Recall that AWAMOCHE is based on TruSt and therefore does not consider the $\langle r_x, w_2\rangle$ race in this trace.) At this point, a standard DPOR would start reversing the races among the z accesses. Doing so, however, is wasteful, since reversing races after the blockage will lead to the exploration of more blocked executions.

Instead, AWAMOCHE chooses to reverse the $\langle r_x, w_1\rangle$ race (as this might make the `assume` succeed), and completely drops the races among the z accesses. We call this procedure *in-place revisiting* (denoted by $\overset{ir}{\rightsquigarrow}$ in Fig. 3). Intuitively, ignoring the z races is safe to do as they will have the chance to manifest in the trace where the $\langle r_x, w_1\rangle$ race has been reversed.

Indeed, reversing the $\langle r_x, w_1\rangle$ does make the `assume` succeed, at which point the exploration proceeds in the standard DPOR way. AWAMOCHE explores 2^N traces where the read of x reads 1, and another 2^N where it reads 2. Note that, even though in this example AWAMOCHE explores 2/3 of the traces that standard DPOR explores, as we show in §6 the difference can be exponential.

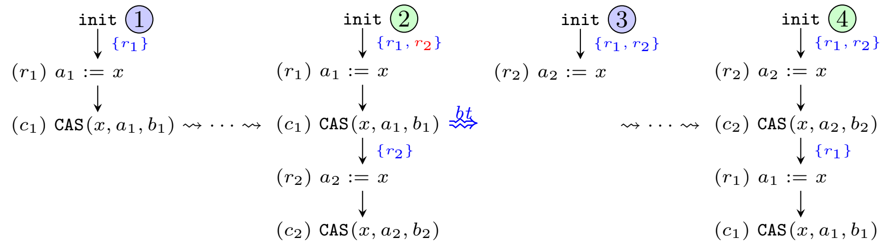

Fig. 5. An AWAMOCHE exploration of the confirmation-CAS example.

Suppose now that we change the `assume`(x) in RW+WW-A-PAR to `assume`(x = 42) so that there is no trace where the assume is satisfied. The key steps of AWAMOCHE's exploration can be seen in Fig. 4. Upon obtaining a full trace, all races to z are ignored and AWAMOCHE revisits r_x in place. Subsequently, as the assume is still not satisfied, AWAMOCHE again revisits r_x in place (trace (2)). At this point, since there are no other races on x it can reverse, AWAMOCHE reverses all the races on z, and finishes the exploration.

In total, AWAMOCHE explores 2^N blocked executions for the updated example, which are all useful. As r_x is reading from the latest write to x in all these executions and the assume statement (corresponding to an await loop) still blocks, each of these executions constitutes a distinct liveness violation.

3.3 Handling Confirmation CASes with Speculative Revisits

In-place revisiting alone suffices to eliminate useless blocking in programs whose assume statements arise only due to await loops. It does not, however, eliminate blocking in *confirmation-CAS loops*. Confirmation-CAS loops consist of a speculative read of some shared variable, followed by a (possibly empty) sequence of local accesses and other reads, and a confirmation CAS that only succeeds if it reads from the same write as the speculative read.

As an example, consider the confirmation-CAS example from §1 and a trace where both reads read the initial value, the CAS of the first thread succeeds, and the CAS of the second thread reads the result of the CAS of the first. Although this trace is blocked and explored by DPOR (since the CAS read of the second thread is reading from the latest, same-location write), it does not constitute an actual liveness violation. In fact, even though the CAS read that blocks does read from the latest, same-location write, the $r := x$ read in the same loop iteration does not. In order for a blocked trace (involving a loop) to be an actual liveness violation, all reads corresponding to a given iteration need to be reading the latest value, and not just one.

To avoid exploring blocked traces altogether for cases likes this, we equip AWAMOCHE with some builtin knowledge about confirmation-CAS loops and treat them specially when reversing races. To see how this is done, we present a run of AWAMOCHE on the confirmation-CAS example of §1 (see Fig. 5).

While building the first full trace (trace (1)), another big difference between Awamoche and standard DPOR algorithms is visible: Awamoche does not maintain backtrack sets for confirmation CASes. Indeed, there is no point in reversing a race involving a confirmation CAS, as such a reversal will make the CAS read from a different write than the speculative read, and hence lead to an `assume` failure.

After obtaining the first full trace (trace (2)), Awamoche initiates a race-detection phase. At this point, the final big difference between Awamoche and previous DPORs is revealed. Awamoche will not reverse races between reads and CASes, but rather between *speculative reads*. (While speculative reads are not technically conflicting events, they conflict with the later confirmation-CASes.) As can be seen in trace (3), Awamoche schedules the speculative read of the second thread before that of the first thread so that it explores the scenario where the confirmation of the second thread succeeds before the one of the first.

Finally, simply by adding the remaining events of the second thread before the ones of the first thread, Awamoche explores the second and final trace of the example (trace (4)), while avoiding having blocked traces altogether.

4 Await-Aware Model Checking Algorithm

Awamoche is based on TruSt [15], a state-of-the-art stateless model checking algorithm that explores *execution graphs* [9], and thus seamlessly supports weak memory models. In what follows, we formally define execution graphs (§ 4.1), and then present Awamoche (§ 4.2).

4.1 Execution Graphs

An execution graph G consists of a set of events (nodes), representing instructions of the program, and a few relations of these events (edges), representing interactions among the instructions.

Definition 1. *An* event, $e \in \mathsf{Event}$*, is either the initialization event* `init`*, or a thread event* $\langle t, i, lab \rangle$ *where* $t \in \mathsf{Tid}$ *is a thread identifier,* $i \in \mathsf{Idx} \triangleq \mathbb{N}$ *is a serial number inside each thread, and* $lab \in \mathsf{Lab}$ *is a label that takes one of the following forms:*

- *Block label:* `B` *representing the blockage of a thread (e.g., due to the condition of an "*`assume`*" statement failing).*
- *Error label:* `error` *representing the violation of some program assertion.*
- *Write label:* $\mathtt{W}^{k_w}(l, v)$ *where* $k_w \subseteq \mathsf{Wattr} \triangleq \{\mathsf{excl}\}$ *denotes special attributes the write may have (i.e., exclusive),* $l \in \mathsf{Loc}$ *is the location accessed, and* $v \in \mathsf{Val}$ *the value written.*
- *Read label:* $\mathtt{R}^{k_r}(l)$ *where* $k_r \subseteq \mathsf{Rattr} \triangleq \{\mathsf{awt}, \mathsf{spec}, \mathsf{excl}\}$ *denotes special attributes the read may have (i.e., await, speculative, exclusive), and* $l \in \mathsf{Loc}$ *is the location accessed. We note that if a read has the* awt *or the* spec *attribute, then it cannot have any other attribute.*

We omit the $\emptyset$ *for read/write labels with no attributes. The functions* `tid`, `idx`, `loc`, *and* `val`, *respectively return the thread identifier, serial number, location, and value of an event, when applicable. We use* R, W, B, *and* `error` *to denote the set of all read, write, block, and error events, respectively, and assume that* $\mathtt{init} \in \mathtt{W}$. *We use superscript and subscripts to further restrict those sets (e.g.,* $\mathtt{W}_l \triangleq \{\mathtt{init}\} \cup \{w \in \mathtt{W} \mid \mathtt{loc}(w) = l\}$*).*

In the definition above, read and write events come with various attributes. Specifically, we encode successful CAS operations and other similar atomic operations, such as fetch-and-add, as two events: an exclusive read followed by an exclusive write (both denoted by the excl attribute). Moreover, we have a spec attribute for speculative reads, and write $\mathtt{R}^{\mathsf{conf}}$ for the corresponding confirmation reads (i.e., the first exclusive, same-location read that is po-after a given $r \in \mathtt{R}^{\mathsf{spec}}$). Finally, we have the awt attribute for reads the outcome of which is tied with an `assume` statement, and write $\mathtt{R}^{\mathsf{blk}}$ for the subset of $\mathtt{R}^{\mathsf{awt}}$ that are reading a value that makes the assume fail (see below).

Definition 2. *An* execution graph G *consists of:*

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

We write $G.\mathtt{R}$ *for the set* $G.\mathtt{E} \cap \mathtt{R}$ *and similarly for other sets. Given two events* $e_1, e_2 \in G.\mathtt{E}$*, we write* $e_1 <_G e_2$ *if* $e_1 \leq_G e_2$ *and* $e_1 \neq e_2$*. We write* $G|_E$ *for the restriction of an execution graph* G *to a set of events* E*, and* $G \setminus E$ *for the graph obtained by removing a set of events* E*.*

Based on the above graph representation, we define $G.\mathtt{po}$, which orders events in the same thread according to their i component, and porf, which is the causal order among the graph events, as follows:

$$G.\mathtt{po} \triangleq \{\langle \mathtt{init}, e\rangle \mid e \in G.\mathtt{E} \setminus \{\mathtt{init}\}\} \cup \{\langle e, e'\rangle \in G.\mathtt{E} \times G.\mathtt{E} \mid \mathtt{tid}(e) = \mathtt{tid}(e') \land \mathtt{idx}(e) < \mathtt{idx}(e')\}$$

$$G.\mathtt{porf} \triangleq (G.\mathtt{po} \cup G.\mathtt{rf})^+$$

The semantics of a program P under a memory model m is the set of execution graphs corresponding to the program that satisfy the consistency predicate of m. Consistency predicates generally constrain the possible choices of co and rf, thereby indirectly constraining the possible final values of memory locations and the values that reads can return.

TruSt (and by extension, AWAMOCHE), assumes some properties on the memory model [15]: porf acyclicity, porf-prefix-closedness, co-maximal-extensibility. Intuitively, extensibility captures the idea that executing a program should never get stuck if a thread has more statements to execute.

Algorithm 1 AWAMOCHE's exploration algorithm

```
1:  procedure VERIFY(P)
2:      VISIT_P(G_∅)
3:  procedure VISIT_P(G)
4:      if ¬consistent_m(G) ∨ ∃b ∈ G.R^blk. ¬maximal(G, b) then return
5:      switch a ← next_P(G) do
6:          G ← G ++ a
7:          case a = ⊥
8:              return "Visited full execution graph G"
9:          case a ∈ error
10:             exit("error")
11:         case a ∈ R^conf
12:             e ← max_po{r ∈ R^spec | tid(r) = tid(a)}
13:             VISIT_P(SetRF(G, a, G.rf(e)))
14:         case a ∈ R \R^conf
15:             for w ∈ G.W_loc(a) do
16:                 if a ∈ G.R^spec ∧ ∃b ∈ G.R^spec. ⟨w, b⟩ ∈ G.rf then
17:                     MAYBEBACKWARDREVISIT_P(SetRF(G, a, w), {b}, a)
18:                 else
19:                     VISIT_P(SetRF(G, a, w))
20:         case a ∈ W
21:             if WWRace(G) then exit("Write-write race")
22:             VISIT_P( IPR (G, a))
23:             Revs ← G.R_loc(a) \ dom(G.porf; [a])
24:             MAYBEBACKWARDREVISIT_P(G, Revs, a)
25:         case _
26:             VISIT_P(G)
```

4.2 Awamoche

Similarly to TruSt, AWAMOCHE verifies a concurrent program P by enumerating all of its consistent execution graphs (see Algorithm 1). In contrast to TruSt, however, AWAMOCHE is *strongly optimal*: it never explores an execution G where there exists some blocked read $r \in G.\mathtt{R}^{\mathsf{blk}}$ that is reading from a non-co-maximal write. In other words, AWAMOCHE only visits graphs that lead to useful executions[2]. In order to be able to do so, AWAMOCHE makes stronger assumptions on the underlying memory model m, namely that there are no write-write races, and that m does not allow porf to contradict co (i.e., that $\mathtt{co} \subseteq \mathtt{porf}$).

Next, we first describe how TruSt works, and then proceed with AWAMOCHE's modifications .

Given a program P, VERIFY visits all consistent execution graphs of P by calling VISIT on the execution graph $G_\emptyset$ containing only the initialization event.

[2] Recall that blocked reads that read from maximal writes *are* useful, as they denote liveness violations.

At each step (Line 4), as long as the current graph remains consistent under the specified memory model m, Visit obtains a new event a via $\mathsf{next}_P(G)$ (Line 5), and extends the current graph G with a (Line 6). We assume that $G{+\!+}a$ adds a to $G.\mathtt{E}$, and also to $G.\mathtt{co}$, in case a is a write. (Recall that $\mathtt{co} \subseteq \mathtt{porf}$ and so a's co-placing is unique.)

If there are no more events to add to the graph, then G is complete, and Visit returns (Line 7). If a denotes an error, then it is reported to the user and verification terminates (Line 9).

If a is a read, Visit needs to examine all possible places where a could read from. To that end, for each same-location write w in G (Line 15), Visit recursively explores the possibility that a reads from w (Line 19). Formally, $\mathsf{SetRF}(G, r, w)$ returns a graph G' that is identical to G except for its rf component:

$$G'.\mathtt{rf} = G.\mathtt{rf} \setminus (G.\mathtt{E} \times \{r\}) \cup \{\langle w, r\rangle\}$$

If a is a write, Visit examines both the case when a is simply added to G (Line 22) and the "*backward-revisit*" cases for each existing same-location read in G that could read from a (Line 5). When a backward-revisits a read r, the resulting graph G' only contains the events that were added before r, or are porf-before a, and updates r to read from a. Since, however, there might be many backward revisits that lead to the exact same graph G', to ensure optimality, G' is visited only when the current graph G forms a *maximal extension* of G'. We do not provide TruSt's definition of maximal extensions here, as Awamoche modifies it to achieve strong optimality.

Let us now move to the parts of Algorithm 1 that are Awamoche-specific.

First, Awamoche discards all graphs where some blocked read is reading non-maximally (Line 4). As explained in §3.2, such reads cannot be revisited and will thus only lead to blocked executions. In addition, to guarantee correctness, Awamoche raises an error if it detects unordered writes (Line 21).

Second, whenever a write event a is added, Awamoche revisits all same-location blocked reads *in place* making them read from a (Line 22) and excluding them from the normal backward-revisit procedure (Line 5). Formally, we define $\mathsf{IPR}(G, a)$ to return a graph G' that is identical to G apart from its rf component:

$$G'.\mathtt{rf} = G.\mathtt{rf} \setminus (G.\mathtt{E} \times G.\mathtt{R}^{\mathsf{blk}}_{\mathtt{loc}(a)}) \cup (\{a\} \times G.\mathtt{R}^{\mathsf{blk}}_{\mathtt{loc}(a)})$$

Third, whenever a confirmation read a is added (Line 11), i.e., an exclusive read that succeeds an unmatched speculative read e, Awamoche only explores the execution where a reads from the same write as e (Line 13): any other write would make the confirmation CAS fail.

Fourth, whenever a speculative read a is added to read from a candidate write w and there is another speculative read b reading from the same write w (Line 16), Awamoche backward-revisits b to read from a. Note that, due to the atomicity of the confirming CASes, there can be at most one other speculative read b reading from w, and so Awamoche revisits it to read from a, making it blocked, so that it get revisited in place when the confirming CAS of a is added to the graph. (To ensure graph well-formedness, we assume that $\mathsf{IPR}(G, b)$ does

Algorithm 2 AWAMOCHE's backward-revisit algorithm

1: **procedure** $\textsc{MaybeBackwardRevisit}_P(G, \mathit{Revs}, a)$
2: **for** $r \in \mathit{Revs}$ **do**
3: $[d_1, \ldots, d_n] \leftarrow \mathsf{sort}_{G_<}(\{e \in G.\mathtt{E} \mid r < e \wedge \langle e, a\rangle \notin G.\mathtt{porf}\})$
4: **if** $\exists G', G''$ such that $G' \overset{r}{\rightsquigarrow} G'' \overset{d_1}{\rightsquigarrow} \cdots \overset{d_n}{\rightsquigarrow} G|_{G.\mathtt{E}\setminus\{a\}}$ and $r \notin G''.\mathtt{R}^{\mathsf{blk}}$ **then**
5: $\textsc{Visit}_P(\mathsf{IPR}(\mathsf{SetRF}(G' +\!\!+ [r, a], r, a), a))$

not modify G when called with a read argument b, and that $\mathsf{SetRF}(G, b, _)$ makes b read from $\bot$, which IPR also considers.)

Finally, similarly to TruSt, AWAMOCHE only performs a backward revisit if G forms a maximal extension, though AWAMOCHE employs a slightly different definition of maximal extensions. AWAMOCHE's backward-revisit algorithm can be seen in Algorithm 2.

Roughly, AWAMOCHE performs a backward revisit from a to r that leads to a graph $\mathsf{IPR}(G_r, a)$ if, starting from G_r without r and a, and adding r and all the deleted events in a co-maximal way (and performing in-place revisits along the way), leads to G. Formally, we write $G_1 \overset{e}{\rightsquigarrow} G_2$ if there exists G'_1 such that $G_2 = \mathsf{IPR}(G'_1, e)$, $G'_1 = G_1 +\!\!+\, e$ and:

$$
\begin{array}{lll}
G'_1.\mathtt{rf} = G_1.\mathtt{rf} \cup \{\langle \mathsf{max}_{G.\mathtt{co}_e}, e\rangle\} & G'_1.\mathtt{co} = G_1.\mathtt{co} & \text{if } e \in \mathtt{R} \\
G'_1.\mathtt{rf} = G_1.\mathtt{rf} & G'_1.\mathtt{co} = G_1.\mathtt{co} \cup \left\{\langle w, e\rangle \middle| w \in G.\mathtt{W}\right\} & \text{if } e \in \mathtt{W} \\
G'_1.\mathtt{rf} = G_1.\mathtt{rf} & G'_1.\mathtt{co} = G_1.\mathtt{co} & \text{otherwise}
\end{array}
$$

We note that, for the special case where $e \in \mathtt{R}^{\mathsf{spec}}$ and there is $e' \in G.\mathtt{R}^{\mathsf{spec}}_{\mathtt{loc}(e)}$ such that e' is not followed by the matching confirmation CAS, we consider $\bot$ as the $\max_{G.\mathtt{co}_e}$. As a final remark, note that, AWAMOCHE modifies $\mathsf{next}_P(G)$ so that (a) after scheduling a speculative read, it keeps scheduling events in the same threads until the respective confirming CAS is added, and (b) it does not schedule events from a thread whose last (speculative) read reads $\bot$. These modifications ensure that the confirmation patterns are added one at a time, and that in-place revisits take place among confirming CASes and speculative reads.

5 Correctness and Optimality

Proving AWAMOCHE correct is non-trivial, as we had to develop a novel proof strategy. In what follows, we first review TruSt's proof argument, show why it is inapplicable for AWAMOCHE. Then, we explain our proof strategy (§ 5.1) and state our completeness and optimality results (§ 5.2).

5.1 Approaches to Correctness

TruSt The proof of TruSt proceeds in a backward manner. Specifically, TruSt's proof is based on a procedure PREV that, given an execution G, recovers the

Fig. 6. TruSt: In-place revisits make it impossible to determine the last step taken

unique "previous" execution G_p that the algorithm must reach in order to visit G. To do so, assuming a left-to-right addition order of events, Prev(G) finds the rightmost porf-maximal event e of G, and decides whether e was added in a non-revisit step, or e is a read that was just revisited by a write event located to its right. If e was added in a non-revisit step, then G_p is simply G without e. Otherwise, Prev obtains G_p from G in the following way: it removes e along with the write w that e reads from, and then iteratively adds the leftmost available event to G in a co-maximal way, until w is about to be added.

TruSt's completeness and optimality are proved using Prev. For the former, one can show that each consistent final execution can reach the initial empty execution through a series of Prev steps, and each of these steps is matched by a forward step of TruSt. For the latter, one can show that each step of TruSt is matched by the (unique) Prev step.

To see why we cannot follow a similar approach for Awamoche, consider the program of Fig. 6, along with one of its executions. We will show that in-place revisits make it impossible to trace the algorithm's last step merely by inspecting the execution. Assuming a left-to-right addition order, Awamoche will reach this execution as follows: it first adds R(x), R(y) and W(x, 1) (notice that at this point the first read is blocked), then in-place revisit R(x), and finally add W(y, 1) and backward-revisit R(y). This last revisit, however, creates a problem: TruSt's proof assumes that a backward revisit $\langle r, w\rangle$ implies that w is located at the right of r, which is clearly not the case here. The fact that in Awamoche backward revisits can happen in both directions, makes it impossible to trace the algorithm's last step simply by inspecting an execution.

Awamoche In contrast to TruSt, Awamoche's proof proceeds in a forward fashion. For each consistent final execution G_f we show 1. which steps are taken by the algorithm in order to reach G_f, and 2. that these are the only possible ones that lead to G_f. To do so, we first define a notion of a *prefix*: we say that an execution G is a prefix of G' (written $G \sqsubseteq G'$), if G' can be reached from G with a series of *operational steps*. In turn, we define an operational step to be a step that the algorithm may take in the non-revisit case (without demanding it is the one actually taken by the algorithm), that may perform in-place revisits as well.

Using this notion of prefixes, our proof defines a procedure Succs that, given a consistent execution G_f and an execution G produced by the algorithm such that $G \sqsubseteq G_f$, Succs returns the minimal sequence of algorithm steps that reach

some execution G' for which it is $G \sqsubseteq G' \sqsubseteq G_f$. Concretely, if $\mathsf{next}_P(G)$ can be added to G such that the resulting execution G' is a prefix of G_f, SUCCS returns this addition step. Otherwise, $\mathsf{next}_P(G)$ is a read event r that must be first revisited by an event e in order to reach an execution that is a prefix of G_f. SUCCS then returns the sequence of algorithm steps that reach the execution resulting from extending G with the porf-prefix of e and setting r to read from e (or from $\bot$, if e is a speculative read). Both completeness and optimality follow from SUCCS's properties, as well as from the observation that every consistent final execution can be reached by a series of operational steps.

5.2 Awamoche: Completeness, Optimality, and Strong Optimality

Before stating our results, we first formally define useful executions. Recall that these are executions where all blocking reads corresponding to await loops are reading maximally (such executions denote liveness violations), and no confirmation CAS fails.

Definition 3. *A consistent execution G is* useful *if every read in $G.\mathtt{R}^{\mathsf{blk}}$ reads from a $G.$co-maximal write and no confirmation CAS fails.*

Next, we define the class of input programs that satisfy our assumptions.

Definition 4. *A program P is* well-formed *if every speculative read is followed by a confirmation CAS with no write in-between, and all writes to locations accessed by speculative reads write distinct values.*

Completeness and Optimality Completeness guarantees that every useful final execution is explored. AWAMOCHE is complete for well-formed programs that do not exhibit write-write races.

Theorem 1 (Completeness). *Given a well-formed program P,* VERIFY(P) *either detects a write-write race and exits, or visits every useful final execution of P.*

Optimality states that (1) no equivalent final executions are explored, (2) there are no *fruitless* explorations that never lead to a consistent final execution.

Definition 5. *We call an execution G visited by* AWAMOCHE fruitless *if it does not recursively lead to any* VISIT(P, G_f) *call, for any consistent final execution G_f.*

AWAMOCHE is optimal for well-formed programs.

Theorem 2 (Optimality). *Given a well-formed program P (1)* VERIFY(P) *never visits two equivalent final executions, and (2) if* VISIT(P, G) *directly leads to a call to* VISIT(P, G') *with G being fruitless, then* VISIT(P, G') *will not initiate any other* VISIT *calls.*

Observe that in the optimality theorem above, fruitless exploration can lead to an extra VISIT step. The reason for that is the treatment of CASes: the read part of a CAS c can be added so that it reads from the same write as a different (successful) CAS. In such a case, there is no way to consistently add the pending write of c without revisiting, which in turn may not be able to happen due to AWAMOCHE's maximality condition.

Strong Optimality Strong optimality states that, apart from being optimal, only useful executions are visited. AWAMOCHE is strongly-optimal for well-formed programs.

Theorem 3 (Strong Optimality). *Given a well-formed program P,* VERIFY(P, G) *only visits useful executions.*

6 Evaluation

We implemented AWAMOCHE as a tool that verifies C/C++ programs under the RC11 memory model [22]. Similarly to other stateless model checkers, AWAMOCHE works at the level of the LLVM Intermediate Representation (LLVM-IR).

In what follows, we evaluate the effectiveness of AWAMOCHE's key ideas (namely, stale-read annotations, in-place revisiting and speculative revisiting) both individually, and as a whole. To that end, we evaluate AWAMOCHE on a set of benchmarks that both amplify the weaknesses of standard DPOR, as well as demonstrate the applicability of our approach in realistic workloads. In all our tests, we compare AWAMOCHE against a vanilla version of TruSt, a version of TruSt that employs stale-read annotations (TruSt$_{\text{STALE}}$), and a version of TruSt that employs both stale-read annotation and in-place revisiting (TruSt$_{\text{IPR}}$).

Even though there are other stateless model checking tools that can be used to verify C/C++ programs (namely, GENMC [19] and NIDHUGG [1]), we do not compare against them here, as we care about AWAMOCHE's performance compared to TruSt. We only mention in passing that we expect GENMC's performance to be similar to that of TruSt$_{\text{STALE}}$ (as its implementation incorporates various optimizations for `assume` statements), and NIDHUGG's similar to TruSt$_{\text{IPR}}$ (as it employs an optimization with a similar effect to in-place revisiting [14]). We also note that comparing with NIDHUGG is difficult since it operates under a different memory model, and does not transform the same types of loops to `assume` statements as AWAMOCHE (also see §7).

We draw two major conclusions from our evaluation. First, AWAMOCHE's optimization yields exponential performance benefits compared to standard DPOR approaches. Second, these benefits do not only apply to small synthetic benchmarks, but also extend to realistic concurrent data structures.

Experimental Setup We conducted all experiments on a Dell PowerEdge M620 blade system, running a custom Debian-based distribution, with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz), and 256GB of RAM. We used LLVM

11.0.1 for AWAMOCHE. Unless explicitly noted otherwise, all reported times are in seconds. We set a timeout limit of 30 minutes.

Table 1. Synthetic benchmarks

		TruSt		$\text{TruSt}_{\text{STALE}}$		$\text{TruSt}_{\text{IPR}}$		AWAMOCHE	
	Executions	*Blocked*	*Time*	*Blocked*	*Time*	*Blocked*	*Time*	*Blocked*	*Time*
`orch-run(4)`	1	15	0.01	0	0.01	0	0.01	0	0.01
`orch-run(5)`	1	31	0.01	0	0.01	0	0.01	0	0.01
`orch-run(6)`	1	63	0.01	0	0.01	0	0.01	0	0.01
`wait-workers(4)`	24	96	0.03	96	0.02	0	0.01	0	0.01
`wait-workers(5)`	120	600	0.09	600	0.09	0	0.03	0	0.03
`wait-workers(6)`	720	4320	0.56	4320	0.56	0	0.14	0	0.14
`nr+nw(3,2)`	0	27	0.01	10	0.03	1	0.01	1	0.01
`nr+nw(5,4)`	0	3125	0.1	126	0.03	1	0.01	1	0.01
`nr+nw(6,5)`	0	46 656	1.32	462	0.06	1	0.01	1	0.01
`conf-loop(4)`	24	256	0.04	176	0.03	124	0.03	0	0.01
`conf-loop(5)`	120	3905	0.09	2010	0.10	1185	0.06	0	0.02
`conf-loop(6)`	720	75 156	1.40	26 916	0.96	13 086	0.54	0	0.08

`orch-run`: N threads are spawned and wait to be signaled before they start performing thread-local computations.

`wait-workers`: A worker thread waits for N workers to publish their results before it starts running.

`nr+nw`: A synthetic benchmark where K reader threads wait until a variable written L times by a writer thread satisfies some condition (which cannot be satisfied).

`conf-loop`: N threads perform a confirmation-CAS loop similar to the one of §1.

6.1 Results

Let us first focus on some benchmarks that help us better understand where each of AWAMOCHE's components can be applied (Table 1). Starting with `orch-run`, we see that even though blocked executions greatly outnumber complete executions, stale-reads annotations alone suffice to bring the number of blocked executions down to zero. This, however, is partly due to luck: in `orch-run`, `main()` spawns a number of workers that do not execute until they are signaled by `main()` using a special variable. In turn, because $\text{TruSt}_{\text{STALE}}$ follows a left-to-right scheduling, when DPOR encounters the worker threads, the scenario where they are not signaled is not considered, since it implies reading a stale value.

By contrast, in `wait-workers` and `nr+nw`, stale-reads annotations are insufficient to eliminate blocking. In these benchmarks, some designated threads wait for the rest of the workers to perform some tasks before proceeding. However, it is not guaranteed that these designated threads are going to be always processed after the rest of the threads by DPOR, and thus stale-reads annotations have little to no effect. Employing in-place revisiting, on the other hand, leads to a dramatic performance improvement: the number of blocked executions is effectively eliminated (the single blocked execution in `nr+nw` is a liveness violation).

Table 2. Real-world benchmarks

		TruSt		$\text{TruSt}_{\text{STALE}}$		$\text{TruSt}_{\text{IPR}}$		Awamoche	
	Executions	*Blocked*	*Time*	*Blocked*	*Time*	*Blocked*	*Time*	*Blocked*	*Time*
`mpmc-enq(4)`	576	1084	0.25	710	0.22	532	0.17	0	0.12
`mpmc-enq(5)`	7200	31 325	4.12	16 382	3.27	12 205	2.72	0	1.48
`mpmc-enq(6)`	86 400	730 626	82.28	303 362	51.29	227 766	42.14	0	19.71
`treiber-push(4)`	24	256	0.07	176	0.04	124	0.04	0	0.04
`treiber-push(5)`	120	3905	0.41	2010	0.29	1185	0.19	0	0.05
`treiber-push(6)`	720	75 156	7.49	26 916	3.61	13 086	1.85	0	0.23
`m-enq(4)`	24	124	0.05	124	0.04	124	0.04	0	0.02
`m-enq(5)`	120	1185	0.11	1185	0.14	1185	0.13	0	0.04
`m-enq(6)`	720	13 086	1.04	13 086	1.05	13 086	1.18	0	0.24

`mpmc-enq`: N threads enqueue an item in a multiple-producer multiple-consumer queue.
`treiber-push`: A lock-free stack implementation. N threads are pushing an item.
`m-enq`: A modification of the Michael-Scott queue without the tail pointer. N threads are enqueueing an item.

Analogously to `wait-workers` and `nr+nw`, `conf-loop` demonstrates why in-place revisiting is insufficient when the success of an `assume` does not depend on a single load, but rather on a sequence of actions (as is the case in confirmation loops). As it can be seen, $\text{TruSt}_{\text{IPR}}$ still explores blocked executions, which Awamoche manages to eliminate thanks to speculative revisits.

Moving to the final part of our evaluation, Table 2 demonstrates that the benefits of Awamoche extend to realistic workloads as well. As can be seen from Table 1, none of Awamoche's optimizations is redundant, as they are often all required to eliminate the exploration of blocked executions. Observe, however, that our benchmarks only exercise push or enqueue operations. This is because the respective pop or dequeue operations contain `assume` statements in their confirmation-CAS loops, and therefore cannot be optimized by Awamoche.

7 Related Work

The seminal work of Flanagan and Godefroid [13] has spawned a number of papers on DPOR. Among these, Optimal-DPOR [2] and TruSt [15] stand out, as they provide the first optimal DPOR algorithm, and the first optimal DPOR algorithm with polynomial memory consumption, respectively. TruSt is based on [17] and thus has the extra advantage of being parametric in the choice of the underlying weak memory model.

A lot of works improve on DPOR one way or another. Many techniques introduce coarser equivalence partitionings to combat the state-space explosion problem (e.g., [3, 10, 11, 12, 6, 8, 7]). Other works focus on extending it to weak memory models [1, 4, 5, 24, 17, 20], while others try to leverage particular programming patterns [16, 18, 14]. Kokologiannakis, Ren, and Vafeiadis [18] in particular, deal with transforming spinloops into `assume` statements, the handling of which we optimize in this paper.

Among those, the work that is closest to ours is GODOT [14]. GODOT is an extension to DPOR that has a similar effect to in-place revisiting in the sense that it only explores executions that are either complete, or denote program termination errors. That said, GODOT only works under SC, and cannot handle stale-read annotations or confirmation loops (which are instrumental in scaling the verification of concurrent data structures, as we saw in §6). In addition, GODOT's loop transformation is static (in contrast to AWAMOCHE's, which is dynamic), making it easy to construct examples where GODOT's transformation does not work. Finally, even though GODOT does not impose a "no write-write race" restriction on the input programs, this restriction is trivially satisfied for models like SC or TSO [26]: in such models, it is sound to transform writes to atomic exchange statements that write the value they read, thereby ordering all writes to each location.

8 Conclusion

We presented AWAMOCHE, the first memory-model-agnostic DPOR algorithm that is sound, complete, and strongly optimal for programs with await and confirmation-CAS loops. AWAMOCHE avoids blocked executions that arise due to await loops by revisiting blocking reads in-place, and deals with confirmation-CAS loops by also considering revisits whenever two speculative reads read from the same write.

As our theoretical and experimental results demonstrate, AWAMOCHE yields exponential benefits over the current state-of-the-art. Yet, it does not support certain more advanced patterns commonly appearing in concurrent programs, the handling of which we leave as future work. Examples of such patterns include confirmation-CAS loops with `assume` statements between the speculative and the confirmation reads (such statements may arise due to `break`/`continue` instructions), elimination backoff data structures, and await loops that use CASes instead of plain reads. We also believe that our key ideas for achieving strong optimality in these cases should be applicable in other scenarios as well, such as in programs with mutual exclusion locks or transactions.

Acknowledgments We thank the anonymous reviewers for their 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).

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: TACAS 2015, LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015)
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM, New York, NY, USA (2014)

3. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. 3, 150:1–150:29 (2019)
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: CAV 2016, LNCS, vol. 9780, pp. 134–156. Springer, Heidelberg (2016)
5. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. 2(OOPSLA), 135:1–135:29 (2018)
6. Agarwal, P., Chatterjee, K., Pathak, S., Pavlogiannis, A., Toman, V.: Stateless Model Checking Under a Reads-Value-From Equivalence. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, pp. 341–366. Springer International Publishing, Cham (2021)
7. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, pp. 526–543. Springer International Publishing, Cham (2017)
8. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, pp. 392–410. Springer International Publishing, Cham (2018)
9. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014)
10. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: TACAS 2018, LNCS, vol. 10806, pp. 229–248. Springer, Heidelberg (2018)
11. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM Program. Lang. 2(POPL), 31:1–31:30 (2017)
12. Chatterjee, K., Pavlogiannis, A., Toman, V.: Value-Centric Dynamic Partial Order Reduction. Proc. ACM Program. Lang. 3(OOPSLA) (2019)
13. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121. ACM, New York, NY, USA (2005)
14. Jonsson, B., Lång, M., Sagonas, K.: Awaiting for Godot: Stateless Model Checking that Avoids Executions where Nothing Happens. In: FMCAD 2022, pp. 163–172. TU Wien Academic Press (2022)
15. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. 6(POPL) (2022)
16. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. Proc. ACM Program. Lang. 3(OOPSLA) (2019)
17. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: PLDI 2019, ACM, New York, NY, USA (2019)
18. Kokologiannakis, M., Ren, X., Vafeiadis, V.: Dynamic Partial Order Reductions for Spinloops. In: FMCAD 2021, pp. 163–172. IEEE (2021)
19. Kokologiannakis, M., Vafeiadis, V.: GenMC: A model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, LNCS, vol. 12759, pp. 427–440. Springer, Heidelberg (2021)
20. Kokologiannakis, M., Vafeiadis, V.: HMC: Model checking for hardware memory models. In: ASPLOS 2020, ASPLOS '20, pp. 1157–1171. ACM, Lausanne, Switzerland (2020)

21. Lahav, O., Namakonov, E., Oberhauser, J., Podkopaev, A., Vafeiadis, V.: Making Weak Memory Models Fair. Proc. ACM Program. Lang. 5(OOPSLA) (2021)
22. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632. ACM, Barcelona, Spain (2017)
23. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9), 690–691 (1979)
24. Norris, B., Demsky, B.: CDSChecker: Checking concurrent data structures written with C/C++ atomics. In: OOPSLA 2013, pp. 131–150. ACM (2013)
25. Oberhauser, J., Chehab, R.L.d.L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., Vafeiadis, V.: VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In: ASPLOS 2021, pp. 530–545. ACM, Virtual, USA (2021)
26. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs 2009, pp. 391–407. Springer, Munich, Germany (2009)